GIT-CC-95-38
FAILURE-DRIVEN LEARNING AS MODEL-BASED SELF-REDESIGN

A THESIS
Presented to
The Academic Faculty

By

Eleni Stroulia

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in Information and Computer Science

Georgia Institute of Technology
December, 1994

DEDICATION

— To my parents –
— To the memory of Grandma Lenaki —

ACKNOWLEDGMENTS

This thesis would not have been possible without the continuous advice and guidance of my advisor, Dr. Ashok Goel. Ashok inspired my decision to work in the field of Artificial Intelligence, played a fundamental role in the formulation of the topic of this thesis, and shaped this research with his vision, his sharp thought, his scientific standards, and his resolution not to let me take shortcuts.

I am also indebted to the other members of my committee, Drs. Ron Arkin, Janet Kolodner, Ashwin Ram and Tony Simon, who were always available to listen to my often ill-formed ideas, and who, with their pointed questions, pushed me to clarify them. Especially, I would like to thank Janet for her very detailed reading of the almost-final version of the dissertation, that made the final version so much better.

All the members of the AI group, in its many instantiations through the years, also deserve my gratitude. Without our discussions many of my ideas wouldn't have taken the shape they have in this thesis, and without their moral support and their conviction that "there is light at the end of the tunnel" my graduate life would have been much poorer. I would like to mention especially Allyana Ziolko, our secretary, for her open-door policy and her warm personality, Erika Rogers for her endless "joie de vivre" and wit, Justin Peterson for his eagerness to "fight" on the issues and his insistence on keeping up with the evolution of style in the outside world, Kavi Mahesh for his distinctive style of humor, the AAAI-93 robotics-competition team for one of the most fascinating conferences I have ever attended, Paul Rowland and Khaled Ali for their help with the Reflecs experiment, Murali Shankar for never seeing the glass half empty, Linda Wills for her always insightful comments, and Mimi Recker for her optimism and her command of statistics. Last but not least, I would like to thank my office-mate and friend, Sam Bhatta, who has helped me think my way out of many frustrating research corners.

The Hellenic Club of Atlanta, and especially Spiros Branis and Spiros Liolis, helped me maintain a real life and, even more importantly, discover the best cafes in Atlanta. My friends, Dwra Farmaki and Penny Sotiris kept Greece alive in their mails, and for that I am grateful. But I owe the deepest thanks to my parents who much as they feared it, they never restrained me from going away from home, and who throughout these years they never seized to believe that I could finish and to hope that I would.

But most of all I want to thank my husband, Yiannis Nikolaidis. This journey would have never started, and would have not been the exciting adventure that it was, had he not been there with me, to keep life in perspective, to make the effort so much easier, to make it all better.

Contents

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Learning is a competence fundamental to intelligence. Intelligent agents who solve problems in a realistic environment need to learn in order to improve their performance in terms of the quality of the solutions they produce, the efficiency of their problem-solving process, and the class of problems they can solve.

Failures in problem solving signify the need and the opportunity to learn. One way in which an agent may effectively use its failed problem-solving experiences to learn, is by reflection upon its own problem-solving process. To that end, the agent needs an explicit model of its own problem-solving behavior. This work adopts a design stance towards reflective, failure-driven, learning. This stance gives rise to a specific computational model which is based on three key ideas: (i) agents can be viewed as abstract devices; (ii) their problem solving can be understood in terms of structure-behavior-function (SBF) models; finally, (iii) failure-driven learning can be viewed as a model-based redesign process, in which the agent uses its comprehension of its own problem solving to repair itself. When the agent fails, it uses feedback from the world, and the trace of the failed process, to search through this model and identify the cause(s) of its failure. Then, it proceeds to repair its problem solving, in order not to fail again for the same reason.

This theory of reflective learning has been implemented in a fully operational system, AUTOGNOSTIC. AUTOGNOSTIC is like a "shell" in that it provides the SBF language for specifying a problem solver, and the inference mechanism for monitoring this problem-solver's reasoning, assigning blame when it fails, and repairing it appropriately. Three different systems have been modeled in AUTOGNOSTIC's SBF language: ROUTER, a path planning system, KRITIK2, a design system, and an autonomous, reactive agent implemented in the AuRA architecture. Extensive experiments conducted with AUTOGNOSTIC demonstrate the generality and the effectiveness of its learning process.

CHAPTER I

INTRODUCTION AND OVERVIEW

> "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively next time." Why should machines learn? H. A. Simon

Learning is a multi-faceted competence, fundamental to intelligent behavior. Intelligent agents learn, for example, by being told (rote learning), by observing an expert performing a task (apprentice learning), and from their own experiences in solving problems in the world (experience-based learning). Failures constitute an especially interesting class of situations in which experience-based learning occurs because they unmistakably signify the need and the opportunity to learn. When an agent fails, it needs to learn to recover from the particular failure, and, it also has the opportunity to eliminate the causes of this failure so that it does not fail for the same reason in the future. This thesis investigates this last type of learning, namely, learning from failures in problem-solving performance.

1.1 The Research Problem

Failure-driven learning is triggered by deficiencies in the agent's problem-solving performance. It has the goal of improving the agent's performance by adapting its knowledge and/or process so that the causes of these deficiencies are eliminated. There are several dimensions to the problem of failure-driven learning which make it an especially hard and interesting problem. In general,

1. the complexity of the agent's problem-solving process may vary,
2. there are several dimensions along which the problem-solving performance can be improved,
3. there are several stages in the agent's problem solving during which a failure may be detected,
4. there are several types of feedback that can be provided to the agent,
5. there are several types of errors that can cause the agent to fail, which, in general, imply different types of modifications, that is, there are several types of learning tasks,
6. there are several types of learning strategies that can potentially be used to mediate the mapping from the agent's failure to its effective and consistent adaptation, and finally
7. each adaptation of the agent's knowledge or process may have undesired affects that can lead to inconsistencies in its overall problem-solving behavior.

1.1.1 Complexity of Problem Solving

The complexity of the failure-driven learning processes increases as the complexity of the problem-solving processes increases. When the agent employs only one problem-solving strategy, it is easier to identify the causes of the problem-solving failure, and in addition, there are fewer ways in which problem solving can be modified.

Many learning systems assume that the problem solver has one, specific, problem-solving strategy. Thus, they limit the learning problem they address, because they propose learning strategies specifically tailored to modify and improve the specific problem-solving strategy. For example, Teiresias [Davis 1977] assumes a particular kind of control strategy, namely backward chaining of production rules, and proposes only one kind of modifications specific to this process, namely refinement of the knowledge base with new rules. Similarly, Lex [Mitchell *et al.* 1981] assumes that problem solving is accomplished through repeated operator application, and the modifications that its learning process proposes are always refinements of the system's operator-selection heuristics.

In general, however, an intelligent agent may employ a diverse set of reasoning strategies. In the case of multi- strategy problem solving, the learning problem becomes more intricate. A single type of modifications does not suffice any more. Instead, what is needed is to analyze the different mechanisms that give rise to the different problem-solving strategies in terms of common language, and to develop an integrated learning process able to identify deficiencies and to propose and effectuate modifications at that level of analysis. This approach gives rise to the following three requirements:

1. a content theory of problem solving, expressive enough to describe a range of problem-solving strategies,
2. an analysis of the types of learning tasks, that is, the types of causes of failures that can occur in problem solving, and
3. an analysis of the types of learning strategies, that is, methods for adapting problem solving, that can potentially be used to eliminate these causes.

1.1.2 Dimensions of Performance Improvement

Simon [1981] characterizes learning as an adaptation mechanism with the goal of improving the system's efficiency and effectiveness on some class of problems. To improve the system's efficiency means to enable the system to solve its problems with increasingly fewer computational resources such as processing time and memory space. To improve the system's effectiveness means to improve the quality of the solutions that the system produces. If, for each problem there is a space of acceptable solutions, then improving the solution quality means narrowing the space of solutions that the system actually produces so that they meet some set of solution- quality criteria. In general however, there is another dimension along which the performance of a system may improve, namely, it may incrementally extend the population of problems it can solve.

> In general, there are three dimensions for improvement of an agent's problem-solving performance which constitute the "function" of learning in the context of an intelligent system:
>
> 1. improvement in the efficiency of the problem-solving process,
> 2. improvement in the quality of the produced solutions, and
> 3. growth of the population of problems that the system can solve.

To date, there have been several computational theories accounting for the improvement of the problem-solving efficiency, e.g., [Anderson, 1982, Carbonell *et al.* 1989, Laird *et al.* 1986, Mitchell *et al.* 1981, Mitchell *et al.* 1989] and several others accounting for the increase of the set of problems that the agent can address, e.g., [Bhatta 1995, Hammond 1989]. Relatively little work has been done on developing theories that account for improvement of the agent's solutions ([Perez 1994] is one example of such work). Finally, there has been even less work that accounts for all three types of performance improvement.

Note that a learning strategy necessary for one type of performance improvement may not suffice for another type. For example, in order to improve its efficiency, the system may compile a sequence of reasoning steps into a single step. However, the ability for such compilation is not, in general, sufficient to enable the system to produce better solutions, or to solve more problems. For example, in order to produce better solutions, the system would potentially have to modify its reasoning steps, so that they "search" for solutions of the desired quality. To solve more problems, the system would potentially have to expand its domain knowledge so that it covers solutions to a larger class of problems.

> Thus, in general, the more the dimensions along which the agent's performance needs to be improved, the wider the range of learning tasks that it needs to address, and the wider the range of the learning strategies it needs to use.

1.1.3 Failure Detection

In principle, there are several ways in which an agent can recognize its failure to successfully solve a particular problem. For example, while solving the problem, the agent may reach a state from which it cannot make any progress. Alternatively, upon producing a solution, the agent may test its solution in the real world, or it may present it to an expert, and find that the solution does not work.

> In general, failures can be detected
>
> 1. either, during problem solving by the agent itself,
> 2. or, after the completion of problem solving, with the help of an external evaluator.

In the latter case the agent relies on a mechanism separate from its own problem solving to recognize its failure. For example, it assumes the existence of a distinct execution/simulation process [Hammond 1989], or an expert [Davis 1977] to evaluate its solutions.

In order to be able to recognize errors in its own problem solving, the agent needs to understand how its problem- solving strategy is supposed to work. In other words, it has to know at an abstract level, i.e., independently of any particular problem, what constitutes a correct problem-solving process. Understanding what constitutes a correct problem-solving process, in turn, requires two different types of knowledge:

1. a specification of the correct behavior of each reasoning step (i.e., each type of inference) in the process, and
2. a specification of how the individual reasoning steps are combined to accomplish the overall tasks of the agent.

These two types of knowledge would enable the agent to monitor its problem solving, to evaluate whether or not the sequence of its reasoning steps, as a whole, makes progress towards accomplishing its tasks, and to evaluate whether or not the intermediate results of its reasoning are correct.

1.1.4 Types of Feedback

Even if an agent is able to detect its own failures autonomously, it cannot always determine the causes of these failures based on the trace of its reasoning and the symptoms of its failure alone. Thus, usually it may require some feedback from its environment that will enable it to identify the cause of its failure, and to repair itself.

The feedback that the environment provides can be of many types. For example, the feedback may simply inform the agent whether the solution it produced succeeded or failed, e.g., [Samuel 1959]. In some cases, the feedback may consist of a complete trace of the execution of the agent's solution, localizing the failure to some step of this trace, e.g., [Hammond 1989, Sussman 1975]. In others, an expert may provide another preferred solution along with a complete trace of how this solution could be produced [Perez 1994]. Alternatively, the feedback may only specify the alternative preferred solution, e.g., [Davis 1977].

It is important to note here that, in this last case, the feedback usually communicates to the agent a particular criterion desired of its problem-solving behavior, as exemplified by this preferred solution. Since, in principle, there may be several different properties desired of the agent's problem-solving performance, the feedback that the agent receives after the completion of its problem solving may be of several different kinds.

The feedback, in the form of a preferred solution, could be indicative

1. of a particular kind of solution quality, or
2. of a particular kind of problem-solving process quality.

1.1.5 The Learning Tasks

In general, there are two types of errors that can cause the problem-solving process to fail:

1. the problem-solver's domain knowledge may be incorrect, or incomplete, or inappropriately represented or organized, or
2. the problem-solver's functional architecture may be incorrect, so that although it may have sufficient knowledge available to it, it does not use it appropriately. For example, the agent may use an inappropriate strategy for solving a particular problem, in spite of having another strategy that would have been successful. Or alternatively, it may ignore some piece of knowledge pertinent to the problem at hand, because it does not even know that this type of knowledge is relevant.

If the environment in which the system operates is static, then the designer of the learning method can realistically assume that the system begins with a complete, correct, and perfectly organized body of domain knowledge, and that errors of the first kind will never occur. A typical example of this approach is Prodigy [Carbonell *et al.* 1989]. Similarly, if there is no variance in the types of problems that the system is presented with, that is, no new constraints or requirements are imposed on the system's problem solving throughout its life, then the designer of the learning method can realistically assume that the system begins with a perfect functional architecture, and therefore errors of the second kind will never occur. A typical example of this approach is Teiresias [Davis 1977]. A weaker version of the same assumption is that the system begins with a complete and correct set of problem-solving elements but without perfect organization of these elements. Then, the learning process becomes responsible for appropriately reorganizing these elements [Carbonell *et al.* 1989, Mitchell *et al.* 1981]. Theories adopting this assumption are able to impose some additional control among the problem-solving elements, but they cannot change the set of these elements and they assume that the problem-solver's domain knowledge is correct.

Neither of the above assumptions is, in general, valid in realistic task environments. More often than not, the state of the system's external environment may change, that is, new objects may appear and the relations among existing objects may change. Furthermore, new constraints may be imposed to the system's problem solving or to the solutions it produces, and thus, the nature of the problems it has to solve may vary. As a result, learning cannot be limited only to acquisition of domain knowledge and refinement of the control flow between the system's functional elements. The learning method must also be able to modify the existing functional elements and to introduce new ones. Such modifications can enable the system to take into account the new environmental

constraints and to tailor its solutions to meet the new requirements imposed on its performance. Thus, the system must be able to recognize need for, and to perform modifications that enable it "to pay attention" to the evolving task and domain, and its opportunities and requirements.

> The system has to be able to autonomously decide what it needs to learn, and set up its own learning tasks. The learning tasks that it has to be able to accommodate are, in principle, of the following types:
>
> 1. (a) To enable the problem solver to keep up with changes in its environment, some learning tasks pertain to the *acquisition of new domain knowledge*.
>
>   (b) To enable effective access to the problem-solver's knowledge, some learning tasks pertain to the *reorganization of its domain knowledge*.
>
> 2. To enable the problem solver to address variations of the original type of problems it was designed for, some learning tasks pertain to the
>
>   (a) *modification of the original elements* of the problem-solver's functional architecture,
>
>   (b) *introduction of new elements*, and
>
>   (c) *reorganization of these elements*.

Here, it must be noted that, the latter kind of adaptations is especially hard. The first reason is because they imply the ability to recognize the need for a new element in the system's functional architecture. A second reason is because they require that a whole new set of information and control interactions be established so that the new element actually contributes to the problem-solving process without compromising its overall consistency. To postulate a new functional element, the agent has to be able to specify a new class of inferences that the problem solver should draw in the course of its reasoning, so that the overall process exhibits the behaviors desired of it. To effectively and consistently integrate the new element in the problem-solving architecture, the agent has to understand the information and control inter-dependencies among its current elements, so that it can modify them without disturbing their composition.

1.1.6 Learning Strategy and Methods

In general, failure-driven learning strategies use the trace of the failed problem-solving episode to identify the cause of the failure, i.e., to identify which problem-solving step(s) led to the failure. This is the well-known problem of blame assignment, that is the identification of the cause of the failure. Earlier AI research on blame assignment has used an external oracle to evaluate each individual step of the problem-solving process until it has identified an erroneous one [Davis 1977, Davis 1980]. Alternatively, the system may exhaustively search its problem space for alternative reasoning paths, and when it finds a successful one, then it can directly compare the failed trace with the successful alternative [Mitchell *et al.* 1981]. Finally, the system may have a set of pre- compiled patterns of erroneous interactions among its own problem-solving steps, and it can identify the cause of each particular failure by identifying instances of these patterns in the trace [Carbonell *et al.* 1989, Ram and Cox 1994, Sussman 1975].

Each one of these approaches to blame assignment has its own merits and disadvantages. The assumption of a resident expert, evaluating the system's performance at a very fine level of detail, is often impossible to meet. Exhaustive exploration of alternative reasoning paths is very costly, especially in the case of multi-strategy problem solvers. Furthermore, the assignment of blame by comparing the incorrect problem-solving trace with a correct one is too narrow a conceptualization

of the blame-assignment task. There may be several correct alternatives, and, in such cases not all differences between the erroneous trace and the correct one are "errors". Finally, the exhaustive enumeration of all possible patterns of errors leaves open the issue of the language in which these patterns should be expressed.

Although the trace defines the space of possible causes for the failure, it does not provide any help in localizing the cause of the failure, or in deciding what precisely the system needs to learn. Traces can be extremely large, especially in the case of complex problems on which the system may spend a lot of effort. In principle, the learning method should investigate the possibility that any of the steps mentioned in the trace could be wrong, since it does not have any basis on which to acquit any of them. In order to avoid this potentially very expensive search, trace-based learning methods have to assume a single type of cause of failure that co-occurs with particular patterns of symptoms in the trace. This assumption simplifies the blame-assignment process into a search for instances of the particular problematic patterns. However, it also limits the scope of blame assignment to identifying a restricted set of types of causes of failure.

> A learning strategy, which does not make a priori assumptions about a specific type of cause of the system's failure, needs
>
> 1. to enable the system to potentially recognize the need for adapting its domain knowledge as well as its functional architecture, and
> 2. to enable the system to effectively address learning tasks of both the above kinds.

To meet the first requirement, the system has to have a language for specifying a taxonomy of learning tasks, some of which pertain to the adaptation of its domain knowledge, and some of which pertain to adaptations of its functional architecture. To be able to actually accomplish such diverse tasks, the system needs to be equipped with an array of learning strategies that can eliminate the different types of errors that cause the problem solver to fail.

1.1.7 Maintaining the Consistency of Problem Solving

Another limitation of traces, as the knowledge mediating the adaptation of the problem-solving mechanism, is that they do not contain any information regarding how this adaptation can be performed in a consistent manner. Existing trace-based methods address learning tasks that do not jeopardize the consistency of the overall problem-solving process, such as refining the heuristics for selecting among the problem-solving operators. They cannot even recognize the need for more complex learning tasks, such as revising these operators for example. But even if they were told about such a need, they would be unable to address it based solely on the information provided by the trace of the problem solving. This is because the trace represents only an example of how the system's reasoning steps, i.e., operators, can be combined, and does not give any information on their potential interactions in general. Thus, their modification might lead to inconsistencies that the trace would be unable to predict.

> To the extent that it is possible the learning process has to ensure that any modifications performed to the system do not compromise the consistency of its problem solving.

To be able to reason about the potential consequences of any adaptation and perform them in a way that does not compromise the overall consistency of the problem-solving process, the system has to understand the inter-dependencies among its functional elements.

1.2 The Approach Adopted in this Thesis

To address the above issues in failure-driven learning, this thesis adopts a specific view of problem solving and learning.

The Device View of Problem Solving

> In analogy to the device view of physical systems, intelligent systems are viewed as abstract devices.

Physical devices are teleological artifacts, that is, they have an intended function and an internal mechanism that results in this function. In an analogous way, intelligent systems are teleological in nature, since they deliver solutions to the problems with which they are presented and which they are expected to solve. Physical devices accomplish their intended functions through their internal causal processes. In turn, these causal processes arise from the interactions of the devices' elementary structural components. Much like devices, intelligent systems accomplish their tasks through their internal problem-solving processes that, in turn, arise from the interactions of the design elements in their functional architectures.

The Design View of Failure-Driven Learning

> In analogy to the design of physical devices, failure-driven learning of intelligent systems is viewed as redesign of a malfunctioning device.

Much like devices that often fail to perform the functions intended of them, intelligent systems often fail to accomplish their tasks. When physical devices fail to perform their intended functions, they must be diagnosed and repaired. In an analogous way, when intelligent systems fail, they must identify the cause of their failure and repair themselves in a way that, in the future, they do not fail for the same reason. To take this analogy one step further, the different kinds of a device failures may necessitate a variety of different types of repairs, including simple adjustment of the parameters of its structural elements, reorganization of the inter-connectivity among these structural elements, and even introduction of new functional elements in it. In an analogous way, the different kinds of failures in a system's performance may lead the learning process to modify the system's knowledge, or to reorganize the interactions among its functional elements, or to even introduce new functional elements in its functional architecture.

The Model-Based View of Reflective Failure-Driven Learning

> In analogy to the model-based redesign of physical devices, explicit models of problem solving enable the assignment of blame and the repair of the causes of the system's failures.

This analogy, from physical devices and their redesign to intelligent systems and learning, gives rise to a process model for failure-driven learning inspired by a process model for redesign of physical devices. When designers redesign physical devices, they often use their comprehension of the internal causal processes of the device. This comprehension enables them to identify the potential causes of the deviation of the observable device behaviors from its intended ones, to select a repair that can potentially eliminate these deviations, to perform the repair on the failing device, and finally, to evaluate its effectiveness [Goel and Chandrasekaran 1989]. In an analogous way, learning might benefit from the comprehension of "how the problem solving processes of the intelligent system work". Such comprehension could support the identification of the potential causes for its failure, the selection of a repair that can eliminate it, the actual repair, and the

subsequent verification of whether or not, the system indeed successfully completes its reasoning on the problem that led to its failure before.

This framework gives rise to the following **research hypothesis**: A model that specifies the functional and compositional semantics of how the system's problem solving works enables it to

1. detect its own failures,
2. identify the potential causes for these failures that may they lie in its domain knowledge or in its functional architecture,
3. autonomously set up its own learning tasks, and
4. repair itself in a way that maintains the overall consistency of its problem solving,

in the context of multi-strategy problem solving.

1.3 Reflective Failure-Driven Learning

To investigate the above hypothesis, one has to provide answers to the following two inter-related questions:

1. What should be the *language*, i.e., the vocabulary and the grammar, for expressing the model of the problem solving such that it can enable reflection and learning?
2. What should be the reflective learning *process*, and how can it make use of the model of the system's problem solving, in order to adapt it and improve its performance?

1.3.1 A Language for Modeling Problem Solving

The first issue in a reflective approach to learning is the level at which to model the problem-solving process. Since the process of adapting the system in such an approach is mediated by the model, the model must contain information about all the elements of problem solving that might be necessary to become the target of this adaptation process. On the other hand, it should abstract away all these other aspects that do not have an impact to the problem-solving competence, and thus do not need to be modified.

This thesis adopts Chandrasekaran's task structures as a starting point for developing a framework for modeling problem solving. In this framework, a problem-solving task is specified by the information it takes as input, the information it produces as output, and a description of the "nature" of the transformation it performs between them. A task can be accomplished by one or more methods, each of which decomposes it into a set of simpler subtasks. A method is specified by the subtasks it sets up, the control it exercises over the processing of these subtasks, and the knowledge it uses. The subtasks into which a method decomposes a task can, in turn, be accomplished by other methods, or, if the appropriate knowledge is available, they can be solved directly.

This recursive decomposition of the overall problem-solving task into methods and subtasks makes explicit the functional, and compositional semantics of the system's problem-solving behavior. The information transformation expected of each subtask constitutes the function that this subtask performs in the context of the overall problem-solving process. The information and control flow among the different subtasks of the problem solver constitute the "causal" interactions among the different functional elements of the problem solver. Finally, the decomposition of a task by a particular method into a set of simpler subtasks provides the rules of composition of a set of lower-level functional elements into a higher-level function.

To describe the system's problem-solving task structure in a computational framework, the language of structure- behavior-function (SBF) models is used. SBF models [Goel 1989] were originally developed for modeling physical devices. The SBF language was a natural candidate for expressing the problem-solving task structure because it was designed to capture the analogs of the functional and compositional semantics of problem solving in the functioning of physical devices.

Adapting this language for modeling problem solving, tasks are expressed as transitions between information states. These transitions are annotated by a set of semantic relations that specify how the task output relates to its input, and thus partially describe the information transformation expected of the task (functional semantics). Each transition acts as index to the methods that can be used to accomplish it. Methods are specified as partially ordered sequences of state transitions. These state transitions correspond to the subtasks into which the method decomposes the task to which it is applied. Thus, their sequence specifies in greater detail how the method accomplishes the task for which it is applicable (compositional semantics). Tasks that are solved directly, without further decomposition, index the procedures that accomplish them. The language used to represent task structures is described in detail in Chapter 3.

1.3.2 A Process Model for Reflection

The comprehension of its problem solving in terms of a SBF model enables an intelligent system to improve its performance in the following ways:

1. by specifying a "road map" for problem solving, it enables the system to **monitor** its progress on a specific problem, and to recognize the lack thereof,

2. by specifying "correctness" criteria for the output of each subtask, it enables the system to **assign blame** for its failure, and to thus set up its own learning tasks, and

3. by specifying how the problem-solving subtasks are composed into accomplishing the overall system task, it guides the system to **repair** its own problem solving in a way that maintains its consistency.

Monitoring
While reasoning on a specific problem, the system can use the model of its own problem-solving process to *monitor* its progress. The model enables the system to record which methods were applicable for any given task, which method was actually invoked to decompose the task at hand, in which specific order the resulting subtasks were performed, which methods were invoked for their respective accomplishment, and what were their corresponding results.

The SBF model of its problem solving provides the system with expectations regarding the information states that it goes through as it solves the problem. The abstract specification of the function expected of each subtask, in terms of semantic relations between its input and output information, constitutes a "standard" for the actual information transformation that this subtask should perform in the context of any problem-solving episode. Based on these standards, the system is able to evaluate its progress, and also the lack thereof. In comparison, traditional, non-reflective, systems are able to evaluate their lack of progress only when they reach a state that is not a final state, and from which they cannot proceed any further. For example, while monitoring its problem solving, the system may notice that it has reached an information state that conflicts with the semantics of the task whose output this state is. In such a case, the system may recognize that it is failing to make progress as expected. Alternatively, the system may complete its reasoning, and produce a solution, and subsequently, it may be informed by its environment that another solution would have been preferable. This type of failures constitute yet another kind of opportunity for learning.

Blame Assignment
Once a failure has occurred, the system may use the record of its failed problem-solving episode, and the SBF model of its problem solving to identify the potential causes of its failure and *assign blame* to some element(s) of its functional architecture. The SBF model explicitly specifies what types of information each subtask consumes, how this information is produced in the overall context of a problem-solving episode, and what types of information each subtask contributes to the higher-level subtask, in service of which it is performed. Essentially, the SBF model "organizes" the interactions among the system's elements in terms of recursively nested functional abstractions.

This knowledge enables the system to localize the cause of the failure of the overall problem-solving task to some specific element(s) in the task structure. Had this knowledge not been available, the blame-assignment process would have to consider all possible interactions among all the functional elements of the system.

Repair
The task-structure theory of problem solving gives rise to a taxonomy of learning tasks, i.e., a taxonomy of types of causes of failure. Associated with each one of these different types of failure causes, there is one or more "repair plans", i.e., learning methods specific to their corresponding learning task. Thus, having identified some potential cause for its failure, the system is able to classify the particular cause into this taxonomy, and thus identify the learning method(s) appropriate for remedying the problem at hand. The goal of the *repair* task is to invoke one of these learning methods in the context of the failed problem-solving episode. The modifications that this learning method may bring about may be simple, e.g., integrating a new fact in its domain knowledge, or more complex, e.g., introducing a new task in its task structure. In any case, the SBF model of problem solving enables the repair process to produce a valid task structure by making explicit the inter-dependencies between the different types of information and the different subtasks. Furthermore, the repair task revises the SBF model of the problem solver to faithfully reflect the modified problem solver.

Verification
The modification is not guaranteed to result in an improved problem solver. However, its effectiveness can be *verified* through subsequent problem solving. If the problem that triggered the modification can now be solved and the appropriate solution produced, this is strong evidence that the modification was appropriate. Alternatively, the system may try to invoke another learning method applicable to repairing the cause of the failure, or if the blame-assignment process has suggested alternative potential causes, it might try to remedy one of these alternatives. These processes are described in detail in Chapters 5, 6 and 7.

1.4 An example from AUTOGNOSTIC

AUTOGNOSTIC is a computational system that implements the reflective learning process sketched in the section above. AUTOGNOSTIC itself does not include a particular problem-solving element. Instead, AUTOGNOSTIC can be integrated with a problem solver. When given the SBF model of a problem solver, it can exhibit the reflective behavior discussed above. This section discusses an example of AUTOGNOSTIC's behavior when integrated with a path planner. Henceforth, the term *agent* (or *system*) will be used to refer to the integration of AUTOGNOSTIC with a problem solver (in this particular example, the path planner). The term *problem solver* will be used to refer to aspects of the problem-solving process (in this example, planning). Finally, the term AUTOGNOSTIC will be used to refer to aspects of the reflective learning process.

1.4.1 The Problem

Consider a path planner that has a topological model of the world in which it operates. Each time it is presented with a new problem, it searches its model in a breadth-first manner to produce a path. Let us also suppose that this planner was designed to be used by pedestrians. Thus when presented with a problem, the planner simply uses its knowledge regarding the inter-connectivity among streets to connect the initial to the final location, and ignores the directionality of the streets since it does not matter to its task. What would happen if the paths of this planner were used by a driver?

Since its current planning process does not consider explicitly the directionality of the pathways when it includes a new segment in its current path, this planner is bound to eventually produce an "illegal for driving" path. Clearly, the task that the planner is required to perform in this new

context is not fundamentally different from the task it currently performs. It is however, a "variant" task, since there are some new constraints that apply to it, which the current path-planning process ignores. The question then becomes,

> How can the agent identify the shortcomings of its current process with respect to the new task requirements, and redesign itself so that these new requirements are met?

1.4.2 The Problem Solver

At a very high level of abstraction, the current process of this planner consists of the following steps:

1. The planner initializes its path to begin at the initial problem location.
2. Then, it searches its model to find locations which are immediately reachable from its current path through the pathways on which its current location lies.
3. Next, it generates a set of new possible paths, by expanding its current path with each one of these reachable locations.
4. Then, it proceeds to select one of these possible paths as its current path, and repeats the process until one of its possible paths has reached the destination.

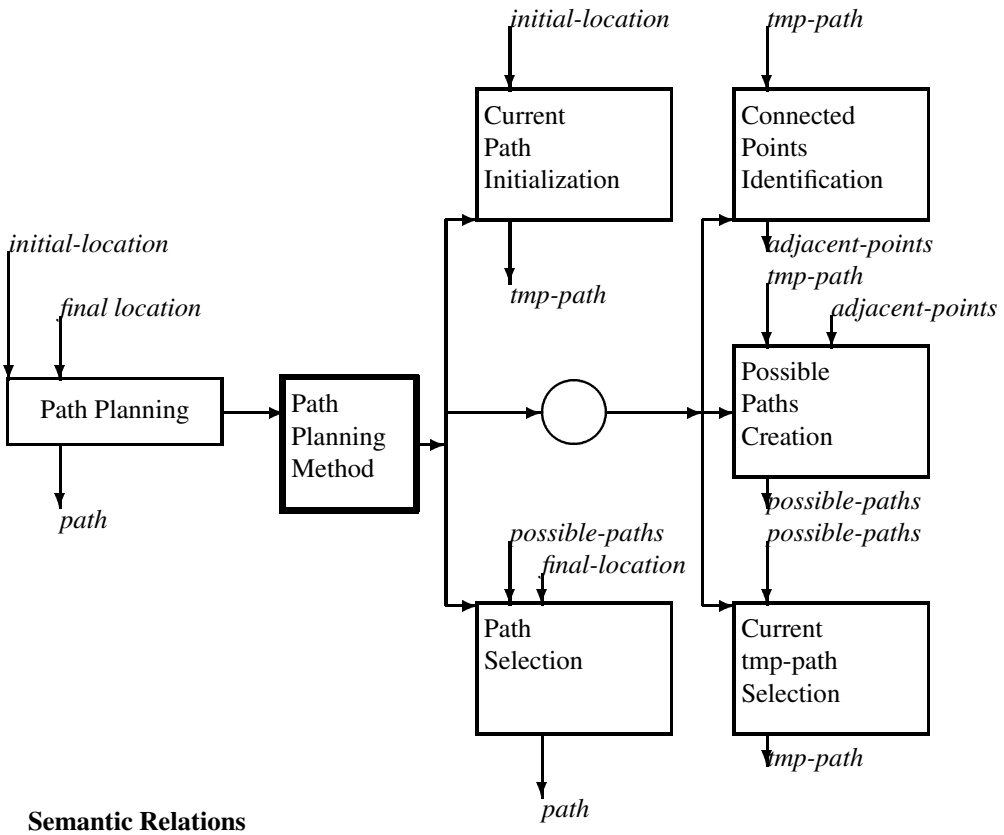
Figure 1.1 diagrammatically depicts the SBF model of this path planner. The single-line boxes depict tasks, the double-line boxes depict methods, and the arrows annotated with italicized labels depict information flow. At the top of Figure 1.1 the planner's task structure is depicted. The overall task of this system is path-planing and it is accomplished by the path-planning method. This method decomposes the overall task into a set of simpler subtasks. The first subtask, current-path initialization, creates a temporary path, tmp-path, which begins at the specified initial-location. After this subtask, as the little circle signifies, the planner repeatedly performs a sequence of three subtasks that modify the temporary path, until the tmp-path reaches the specified final-location. More specifically, the first subtask of the repeated sequence, connected-points identification, identifies the set of locations which are adjacent to the planner's current location, namely the final point of the temporary path. Then, the possible- paths creation subtask extends the temporary path to all these points, and finally, the current tmp-path selection subtask selects one of the possible paths as the current tmp-path. The final subtask path- selection simply establishes the current temporary path as the solution path.

At the bottom of the Figure 1.1 an abstract specification of the planner's domain knowledge is depicted. The path planner understands its environment in terms of street intersections and a set of connections among them. This connectivity relation is the knowledge on which the planner bases its inference about the adjacent points of a location. This is expressed in the semantic relations characterizing the connected-points identification subtask.

1.4.3 AUTOGNOSTIC in action

Let us illustrate AUTOGNOSTIC's reflective learning process in the above scenario. Figure 1.2 depicts a small part of the new domain of the planner, including the legal directions of the pathways.

Monitoring The planner is presented with the problem of going from *(myrtle & cherry)* to *(north & mapple)*, and it invokes its path-planning method to solve it. As the planner reasons about this problem, AUTOGNOSTIC monitors its progress and records its inferences. At each point in the reasoning process, based on the SBF model of the path-planner, AUTOGNOSTIC knows what is the planner's current task, which method it uses to accomplish it, how the current task contributes to



Semantic Relations

Path Planning:

$equal(initial\_location\ \ first(path)) \wedge equal(final\_location\ \ last(path))$

Connected-Points Identification:

$\forall n \in adjacent\_points,\ \ connected(n\ \ last(tmp\_path))$

Possible-Paths Creation:

$\forall p \in possible\_paths,\ \ subpath(p\ \ tmp\_path)$

$\forall p \in possible\_paths,\ \ \exists n \in adjacent\_points,\ \ equal(n\ \ (last(p)))$

Current Tmp-Path Selection:

$\exists p \in possible\_paths,\ \ equal(p\ \ tmp\_path)$

Path Selection:

$\exists p \in possible\_paths,\ \ equal(p\ \ tmp\_path) \wedge equal(last(p)\ \ final\_location)$

Domain Concepts	Attributes	Domain
Intersection:	streets (list-of 2 street)	int-table
Street:		
Path:	nodes (list-of intersection)	
	...	
	final-segment (intersection intersection)	
Domain Relations	Mapping	Table
connected:	(intersection) → (list-of intersection)	connectivity-table
legal-direction	legal-direction: (intersection intersection) → {T F}	traffic-rules-table

Figure 1.1: The SBF model of the path-planner's reasoning.

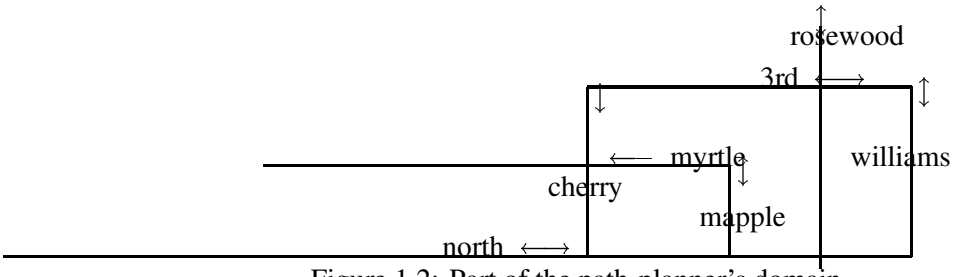


Figure 1.2: Part of the path-planner's domain.

the overall path-planning task, and what other subtasks are still required in order for the overall path-planning task to be accomplished.

Thus, AUTOGNOSTIC records that the current-path initialization subtask initializes the tmp-path to be *((myrtle cherry))*. Subsequently, the planner starts the first repetition of the three-subtask sequence. The connected-points identification subtask produces as *adjacent-points* the set { *(cherry north) (myrtle mapple)* }. The possible-paths creation subtask produces as possible-paths the set { *((myrtle cherry) (cherry north)) ((cherry myrtle) (myrtle mapple))* }. The current tmp-path selection subtask renews the value of *tmp-path* to be an arbitrary path from this set, *((myrtle cherry) (myrtle mapple))*. At the end of the sequence the exit condition of the *loop* control operator is evaluated, and since this condition is not true, i.e., no path has reached the final–location yet, there is another repetition. After a second repetition of the sequence, the value of *tmp-path* has become *((cherry myrtle) (myrtle mapple) (mapple north))*, and the exit condition of the *loop* is true. Thus, the planner performs its last subtask path-selection which produces as the value of the solution, path , *((cherry myrtle) (myrtle mapple) (mapple north))*.

Blame Assignment This path traverses *myrtle* street from *West* to *East*, which is not a legal driving direction, as shown in Figure 1.2. Let us assume that a driver tries to use this path, and notices that it is illegal. Let us further assume, that since it cannot execute the above path, the driver explores the area shown in Figure 1.2 and manages to reach its destination, through the path *((cherry myrtle) (cherry north) (north mapple))*,. Then it communicates this path back to the agent. At this point, AUTOGNOSTIC has to assign the blame for the planner's failure to produce this successful path.

Initially, AUTOGNOSTIC uses its model of planner's domain knowledge to parse the desired solution in order to assimilate all the information it contains. In principle, the planner may have failed to produce the desired solution because it does not have all the domain knowledge pertinent to that solution. For example, the planner might not have produced the preferred path because it did not know about the intersection between *cherry* and *north*. Therefore, the initial blame-assignment step of feedback assimilation aims at identifying whether the desired solution conveys information which is missing from or conflicting with the planner's domain knowledge.

In this example, from the description of the planner's domain concepts in its SBF model, (see bottom of Figure 1.1), AUTOGNOSTIC knows that a path refers to a list of intersections. Thus, given the desired path, AUTOGNOSTIC evaluates whether all the intersections it contains already belong in planner's knowledge of intersections, in the int–table . For this example, this is indeed true. Thus, at this point AUTOGNOSTIC establishes that all the elements of the solution are known to the planner, and thus the cause of its failure must lie somewhere within the process that produced the failed solution. If there were intersections or streets in the desired path which the planner did not know, then AUTOGNOSTIC would infer that a potential cause of the planner's failure was its incomplete knowledge of the world, and it would have suggested the acquisition of the new intersection in the planner's domain knowledge.

If the assimilation step proceeds without errors, i.e., all the information conveyed by the desired solution is already known to the planner, AUTOGNOSTIC tries to identify modifications which could potentially enable the planner to produce the desired solution using the same sequence of reasoning steps that led to the failed solution before. It is possible that the task structure allows the production of several solutions to the same problem, some of which is more desirable than the others, and

the problem solver does not know how to direct its reasoning towards the right kind of solutions. Another possibility is that the planner may occasionally draw incorrect inferences due to errors in its world knowledge, although its underlying problem-solving mechanism is correct. In the case of the former type of error, an appropriate modification could be to introduce some new task in the task structure which would be able to distinguish between the possible alternatives and would steer the problem-solving process towards the correct solution. In the latter case, the appropriate modification could be to modify the world knowledge so that it does not support incorrect inferences.

In this example, based on the semantic relation of the path-selection subtask, (see Figure 1.1), AUTOGNOSTIC realizes that in order for the desired path to be produced in this particular problem-solving episode, the possible-paths set should include the path *((myrtle cherry) (cherry north) (north mapple))* . In order for this path to belong in the possible-paths set, the tmp-path input to the possible-path creation subtask should have been *((cherry myrtle) (myrtle mapple))*. In turn, this tmp-path should have been produced by the subtask current tmp-path selection of the previous repetition of the three-task sequence.

At this point, AUTOGNOSTIC realizes that the subtask current tmp-path selection , when performed for the first time, could have produced any of the two values { *((myrtle cherry) (cherry north)) ((cherry myrtle) (myrtle mapple))* } as a tmp-path , because both these values conform with its semantics. However one of them, i.e., { *((myrtle cherry) (cherry north))* }, is more preferable than the other, because it leads to the overall desired path . At this point, AUTOGNOSTIC recognizes that one potential reason for the planner's failure may be that the current functionality of its current tmp-path selection task is under-specified. That is, the mapping that this task performs from its input, possible-paths , to its output, tmp-path , allows for wrong inferences. Therefore, what is needed to repair the planner is to constrain the functionality of the failing task by refining its input-output mapping.

Repair In order to refine the input-output mapping of the failing current tmp-path selection task, AUTOGNOSTIC has to identify a means for differentiating between the type of tmp-path that is acceptable for this task and the type that is not, as exemplified by the desired value of the tmp-path in this example, *((myrtle cherry) (cherry north))* , and the actual value that the planner produced while planning, *((cherry myrtle) (myrtle mapple))* . Searching in the planner's domain theory, AUTOGNOSTIC discovers, that the legal-direction of the last-segment of the tmp-path could constitute such a criterion. More specifically, AUTOGNOSTIC discovers that the current tmp-path selection should prefer to produce as tmp-path , one path whose *last-segment* traverses a street along one of its *legal-direction*s[1]. Therefore, AUTOGNOSTIC suggests the failing task, current tmp-path selection , should be modified to produce only the "right kind" of paths as tmp-path , i.e., the paths that conform with the above criterion.

Figure 1.3 depicts the SBF model of the planner after it was modified according to the suggestion of the blame-assignment task. The functionality of the old current tmp-path selection subtask has been refined. Its expected correct behavior is characterized by an increased, more selective set of semantics; the output of this task now conforms with the newly found criterion, in addition to the old task semantics. Correspondingly, the procedure that carries out this task has been modified to return the right kind of output.

Verification After having modified the planning process, AUTOGNOSTIC presents the planner with the same problem that the led to the failure in response to which the modification was performed. If the problem is successfully solved, which is the case in the particular example, then the repair is evaluated as successful. This is indeed the case with this example.

1.5 Research Goals Revisited

Let us now define more precisely the research problem addressed in this thesis along the dimensions explored in section 1.1.

[1]Notice that if each time the last segment of the path is legal, the complete path will be legal

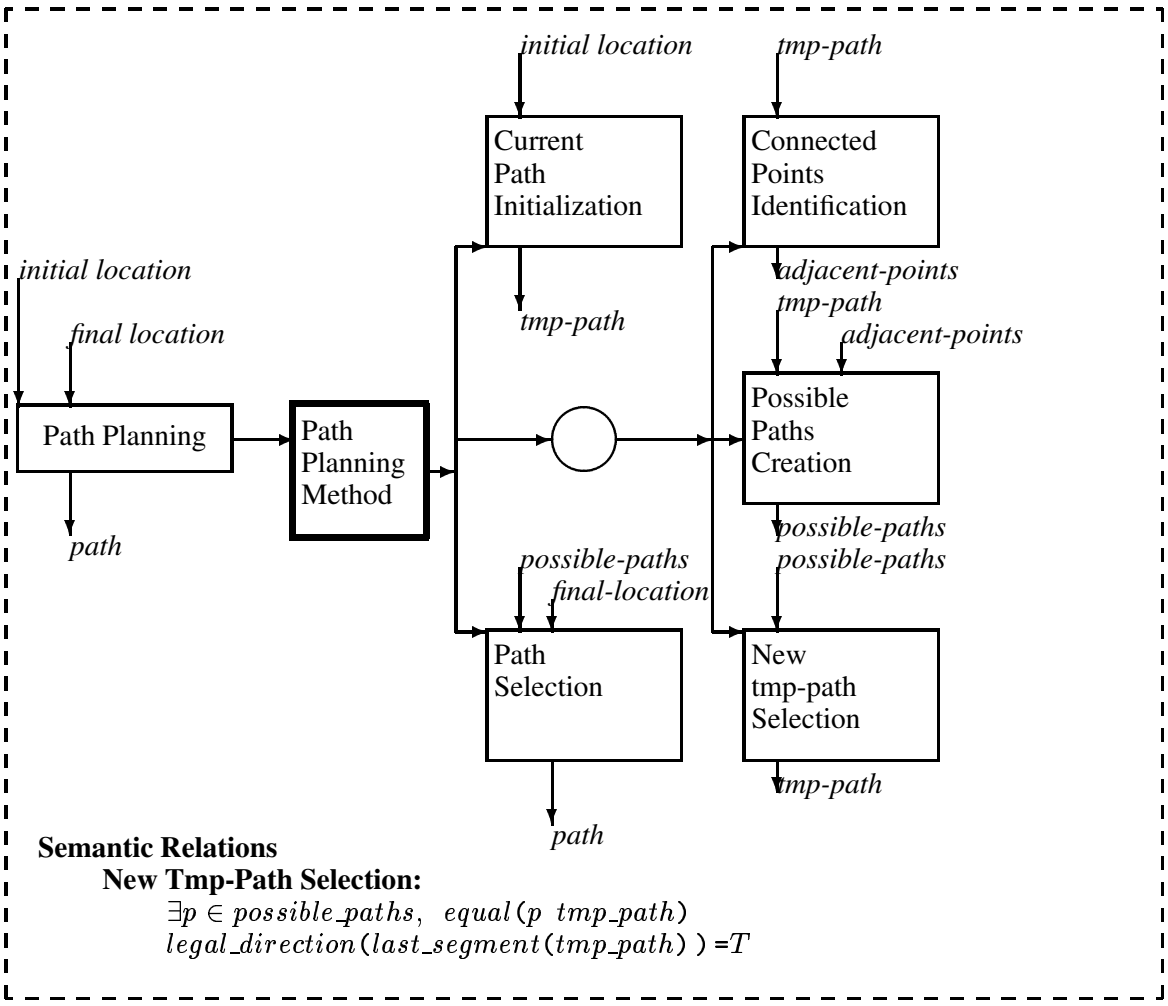


Figure 1.3: The modified SBF model of the path planner.

- This theory of reflective failure-driven learning is intended to be applicable to systems with multiple problem-solving strategies.
- It is intended to account for improvement of the system's performance with respect to three criteria: the quality of the solutions it produces, the efficiency of its problem-solving process, and the population of problems that it can solve.
- The learning process can be triggered by failures that the reflective system itself is able to recognize, based on its comprehension of the nature of the tasks it is designed to accomplish, or by failures that the environment indicates to the system, in terms of alternative solutions preferred to the ones the system actually produced.
- The learning process admits multiple types of causes for the system's performance failures, and therefore, the need of the system to set up for itself a wide class of learning tasks. More specifically,
  - The learning process must be able to acquire new domain knowledge and integrate it in the problem-solver's knowledge base, to reorganize the domain knowledge so that

it can be effectively accessed, and to recognize the insufficiency of the knowledge-representation scheme to express new pieces of knowledge.

– The learning process should be capable of modifying the agent's primitive functional elements, and introducing new ones, when the problem solver is required to accomplish tasks which are variants of the tasks performed by its original reasoning strategies.

- Finally, the learning process should provide learning methods appropriate for accomplishing the above types of learning tasks, which should enable the system to repair its own problem solving without disturbing its overall consistency.

1.6 Evaluation

In general, any theory of learning can be analyzed and evaluated in terms of the following dimensions:

1. Computational feasibility
2. Generality
   - with respect to the class of intelligent systems it applies to, and
   - with respect to the class of learning tasks it addresses
3. Effectiveness
   - with respect to the dimensions along which it improves the system's performance, and
   - with respect to the range of situations in which it does so.
4. Realism

AUTOGNOSTIC is a fully operational, computational system that implements the reflective learning process described above. This demonstrates its computational feasibility.

To evaluate the generality of the SBF language and the reflection process, AUTOGNOSTIC was integrated with three different problem solvers. Two of them are deliberative, knowledge-based problem solvers: ROUTER [Goel *et al.* 1991, Goel and Callantine 1992, Goel *et al.* 1993, Goel *et al.* 1995], a path planner, and KRITIK2 [Bhatta and Goel 1992, Goel 1989, Stroulia *et al.* 1992], a designer which designs physical devices at the conceptual level by adapting the designs of other devices it knows about. The third problem solver was a reactive planner implemented in the context of the AuRA architecture [Arkin 1986]. All these problem solvers were designed and developed independently of this work, and in different research paradigms. Furthermore, the tasks they perform are quite diverse. Finally, the problem-solving theories that they embody are fundamentally different, i.e., deliberative vs. reactive, with a lot of internal knowledge vs. completely without internal representations. The diversity of these three problem solvers suggests that the reflective learning process is applicable to a wide class of problem solvers characterized by the fact that their problem-solving mechanisms consist of a set of identifiable design elements with well-defined functionalities and well-defined interactions between them.

To evaluate the generality of the reflective learning process with respect to the class of learning tasks it addresses, a wide range of different problem scenarios was presented to AUTOGNOSTICONROUTER and AUTOGNOSTICONKRITIK2. These scenarios cover different combinations of failures and feedback, and lead to different learning tasks. Thus, they demonstrate the flexibility of the reflection process and its ability to accomplish a range of learning tasks.

The effectiveness of AUTOGNOSTIC's reflective learning was evaluated in terms of two dimensions: first, with respect to three different performance-improvement criteria, i.e., quality of produced solutions, process efficiency, and class of solvable problems, and second with respect

to the amount of problem-solving experiences that the system had, i.e., learning based on a single problem-solving episode, or incremental learning based on a sequence of problem- solving experiences. Different experiments were conducted for all these six conditions, and in all of these experiments the performance of the problem solver improved. This result demonstrates the effectiveness of AUTOGNOSTIC's learning process.

Finally, to evaluate the degree to which this reflective learning process is realistic, its knowledge assumptions are analyzed from both a cognitive-science and an intelligent-system-design perspective.

1.7 Contributions

This thesis proposes a computational theory of failure-driven learning which demonstrates how an intelligent system can learn and improve its performance. It endows the system with the ability to reflect on its failed problem-solving episodes, guided by a SBF model of how its problem-solving process works. The main contributions of this thesis are the following:

1. A content theory of the types of learning tasks that may arise in failure-driven learning, grounded in the task-structure view of problem solving, and a language for explicitly representing them.

2. A method for monitoring problem solving that enables the system to autonomously recognize its own failures.

3. A method for the general blame-assignment task which

   (a) admits that failures can be caused either by errors in the content, organization, or representation of the system's domain-knowledge, or by errors in the content and organization of the system's functional architecture, and

   (b) enables the system to autonomously set up its own learning tasks.

4. A method for selecting a learning task among the ones potentially relevant to a problem-solving failure, and a method among the ones potentially applicable to the chosen learning task.

5. A array of learning methods that

   (a) are able to address learning tasks of both the above types,

   (b) in a manner that maintains the overall consistency of the problem-solving process.

The learning method assumes the availability of a SBF model of the problem-solving process under reflection. However, it does not assume that this model is complete or correct, and it provides a partial account for how this model can be incrementally improved.

It makes use of a particular kind of feedback, namely, the solution desired of the problem-solving process for a given problem. This feedback is not necessary (the learning method can proceed even without it), however, the range of learning tasks that it can accomplish increases with the existence of such feedback.

This method is applicable to single- as well as multi-strategy problem solvers, as long as the complete set of their design elements can be identified and their respective functionalities can be described. Finally, this learning method results in performance improvement along three dimensions: the set of problems the problem-solver can solve, the quality of solutions it can produce, and the efficiency of its problem-solving process.

1.8 Thesis Outline

Chapter 2 describes the major issues that arise in developing a reflective, failure-driven, learning process, the functionalities of each one of the subtasks of this process, and their respective knowledge needs.

Chapter 3 describes in greater detail the language for modeling problem solving. It discusses the issues involved in deciding which aspects of problem solving should be preserved in its model, and analyzes the task-structure view of problem solving. Further, it specifies AUTOGNOSTIC's SBF language in terms of a context-free grammar and illustrates this language with examples from the ROUTER's SBF model.

Chapter 4 discusses the other two problem solvers with which AUTOGNOSTIC has been integrated KRITIK2 and REFLECS, and describes their SBF models. This chapter also describes in detail the process by which AUTOGNOSTIC is integrated with a problem solver, using its integration with REFLECS as an example, and discusses some implementation issues of the process.

The following three chapters describe in detail the methods developed for each of the three subtasks of the reflective learning process. They are all organized similarly. First, the issues regarding the development of a mechanism for each subtask are presented. Next, the mechanism for each subtask, as developed in AUTOGNOSTIC, is described, along with the relevant algorithms. Next, this mechanism is illustrated with a set of examples sufficient to cover all the aspects of its possible behavior. Finally, each chapter closes with a short discussion summarizing the important aspects of the mechanism. Chapter 5 describes the monitoring subtask. Chapter 6 describes the subtask of blame assignment. Chapter 7 describes the repair subtask.

Chapter 8 walks through a complete problem-solving-and-learning episode with AUTOGNOSTIC and reviews the issues that arise in each step of this process, the inferences that AUTOGNOSTIC makes to resolve these issues and the knowledge that enables it to draw these inferences.

Chapter 9 presents an overall evaluation of the reflection process as a method for learning from failed problem- solving episodes. The computational feasibility, the generality, the effectiveness and the realism of the reflection process implemented in AUTOGNOSTIC are evaluated in the context of the three problem solvers integrated with AUTOGNOSTIC.

Chapter 10 relates the approach adopted in this thesis and its results with other relevant AI research in modeling problem solving, learning and problem solving, and reflection and also with psychological research on reflection.

Finally, chapter 11 draws some conclusions from this research, and outlines directions for future research.

CHAPTER II

A PROCESS MODEL FOR REFLECTIVE PERFORMANCE-DRIVEN LEARNING

The overall task of failure-driven learning takes as input a problem solver that exhibits some deficiencies in its problem-solving behavior, and has as a goal to deliver as output the same problem solver repaired, i.e., with the cause of its performance deficiency eliminated.

In principle, there are several strategies that could potentially be used to address this task. For example, one such strategy would be to assume a particular state in the problem-solver's reasoning where failures are detected, and to use associations to directly map the specific symptoms of the failing state to appropriate repairs. This strategy might be possible under the following conditions. First, failures have to be detected always at the same stage of the problem-solving process. Second, the failing problem solver must be sufficiently simple so that there are only a few possible ways in which it can be adapted. Otherwise the set of necessary failure-to-modification associations may become excessively complex. However, these conditions do not usually hold true. As discussed in Chapter 1, failures may be detected during the problem-solver's reasoning or after its seemingly successful completion; therefore the learning strategy cannot assume a single stage where it can look for failures. Furthermore, problem solvers can be quite complex, and may fail in several different ways to which a variety of adaptations can potentially be performed; therefore the mapping from failures to adaptations becomes increasingly complex.

An alternative strategy, which could address these issues, might be to actively monitor the complete problem-solving process, and to mediate the mapping from failures to adaptations with the identification of the possible causes of the failure. On one hand, monitoring the whole problem-solving process enables the agent to potentially recognize failures whenever they may occur. On the other, establishing an intermediate space between the space of failure symptoms and the space of possible adaptations, limits the complexity of the mapping, because this intermediate space is, in general, smaller than either of the other two. This is because a single cause may give rise to a variety of failures, each one with potentially different symptoms, depending on the problem-solving context. Also, for each cause there may be several potential ways to eliminate it.

Having adopted a *monitoring-identification of the cause of the failure-repair* failure-driven learning strategy, a whole new set of issues arise: first, how to recognize a failure in the problem-solver's reasoning, second how to identify the cause of the failure, and third, how to decide on a repair which can potentially eliminate it. One approach, adopted by several AI systems [Carbonell *et al.* 1989, Davis 1977, Mitchell *et al.* 1981], has been to assume that there exists one single type of error that is responsible for all the problem-solver's failures, and correspondingly, a single type of repair that can be employed to fix it. Under this assumption, the problem becomes to simply to localize an instance of the error in the problem-solver's reasoning, and to instantiate the known repair in that specific context. This approach, however, becomes inapplicable in the case of multi-strategy problem solvers. Multi-strategy problem solvers may suffer from an array of possible types of errors, each of which may be particular to one of their different problem-solving strategies, and may require a different type of repair in order to be eliminated. Therefore, to address the problem of failure-driven learning in the context of multi-strategy problem solving, a learning strategy must meet the following criteria. First, it must be able to identify multiple types of causes of failures in the problem-solving process. Second, it must be able to perform several kinds of adaptations to the failing problem solver. Finally, it must be able to decide which adaptation to perform when.

Finally, it is important to note that, as the adaptation competence of the learning strategy increases, i.e., as the types of modifications it can perform to the failing problem solver become increasingly diverse, the problem of maintaining the consistency of the problem-solving process becomes increasingly hard. Even if the learning strategy has an array of repair methods corresponding to the types of causes of failures it can identify, there is no guarantee that these methods will always be successful. The application of a repair method to one particular cause of failure may have non-local side effects to the problem-solver's reasoning that compromise the consistency and the integrity of the overall problem-solving process. Furthermore, even if a particular repair is successful in eliminating the particular cause it targeted in the context of the failed problem-solving episode, it may give rise to problems which may become evident in later problem-solving episodes. Therefore, a learning strategy should also be able to reason about the affects of the modifications it performs to the integrity of the overall problem-solving process, and to verify their effectiveness.

The above analysis gives rise to the following major subtasks for the overall task of failure-driven learning:

1. monitoring
2. blame assignment,
3. repair, and
4. verification.

However, it does not make any commitments regarding the types of knowledge that these subtasks require. To propose types of knowledge that can enable the accomplishment of these tasks, let us appeal to the analogy of failure-driven learning to model-based redesign of physical devices. Just as a model of "how a physical device is designed to function" enables the localization of the cause of the device failure and its consistent repair when it malfunctions, a model of "how the problem solver is designed to reason" can enable the assignment of blame for the problem-solver's failures and its effective and consistent repair.

The process model that arises from this model-based approach to the issues that arise in failure-driven learning is depicted in Figure 2.1. This figure illustrates the functional architecture of a reflective problem-solving and learning system. Such a system can be viewed as reasoning in two distinct spaces. In the reasoning space, the system uses its domain knowledge to solve the problems it is presented with, i.e., to produce a solution $s_{PS_t}(p)$ for each input problem p. In the meta-reasoning space, the system uses a model of its own problem solving to monitor and adapt itself towards improving its own performance, i.e., towards improving the efficiency of its processing, or improving the quality of the solutions it produces, or increasing the set of problems it can solve. The distinction between these two spaces does not imply a fundamental difference between the mechanisms that give rise to the system's reasoning in these two spaces. Instead, it illustrates a property characteristic of a reflective system, that such a system has knowledge about its problem-solving process above and beyond the knowledge that is necessary for this process to occur.

The following sections of this chapter discuss in greater detail the role that each task plays in the context of the overall failure-driven learning task, as well as the knowledge requirements that the model must fulfill in order for each task to be accomplished.

2.1 Monitoring

The monitoring task takes as input the model of the problem solver, and its actual problem-solving behavior on a given problem p, and has as a goal to detect potential failures of this behavior, and also to produce as output a record of this problem-solving behavior.

The role of the monitoring process in the context of the overall failure-driven learning task is two-fold:

1. to establish the need for learning, by detecting failures of the problem solving, and

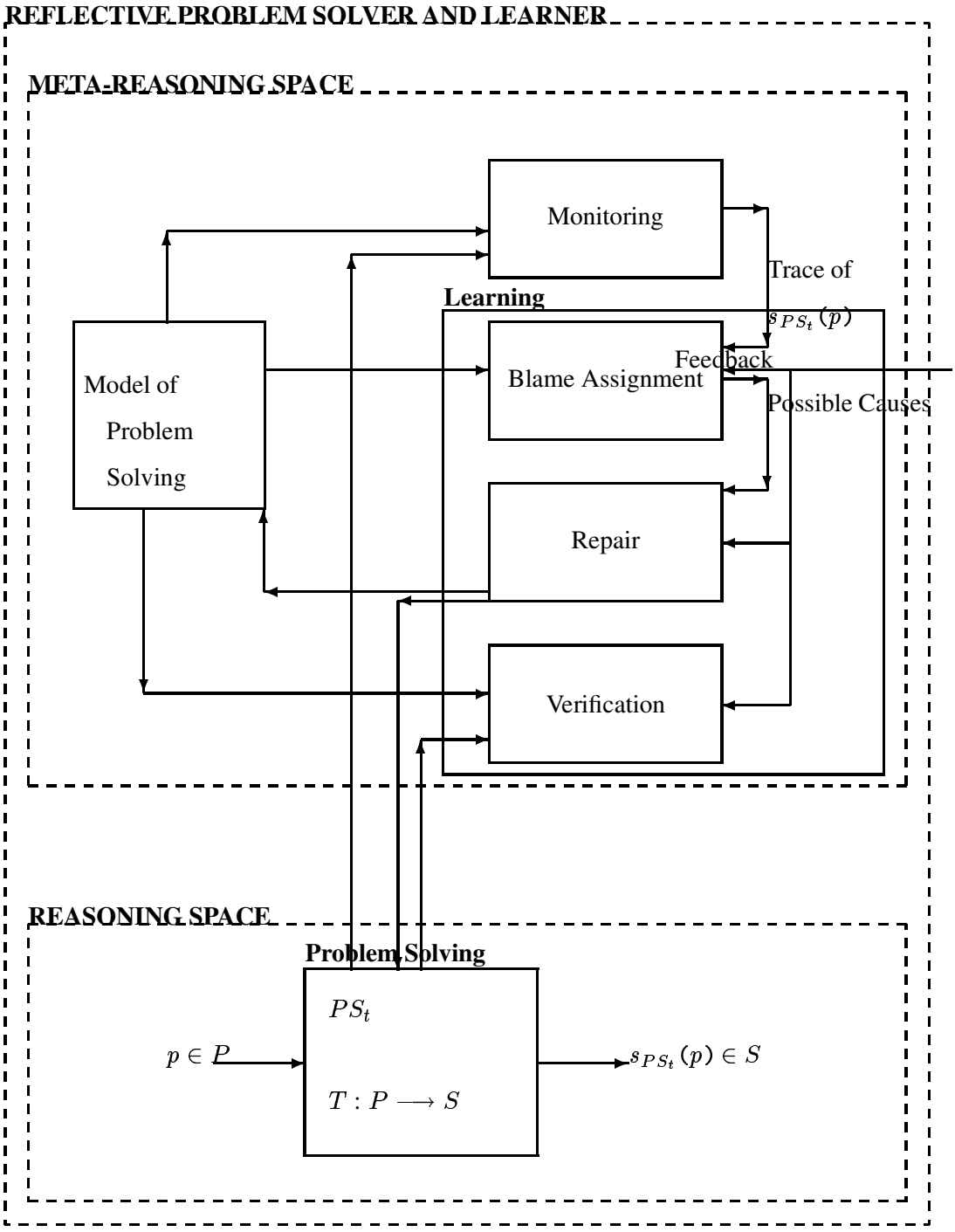


Figure 2.1: The Functional Architecture of a Reflective Problem Solving and Learning System.

2. to set up the information context for learning.

To accomplish the first aspect of its function, the monitoring task needs to recognize when the problem is actually solved, or alternatively, when no progress is being made because a failure has occurred. More specifically, the monitoring task requires some type of knowledge that can enable it to evaluate the success and the failure of the problem solving.

One way in which the problem-solving progress can be evaluated is with respect to its final results. That is, the monitoring task can recognize that the problem solver has successfully solved the problem at hand, when it has reached a state that exactly matches a desired state specified in the problem description. Correspondingly, it can recognize that the problem solver has failed, when the problem-solving process has halted at a "sink" state, that is, a state different from the desired one, from which no further progress can be made. This approach, however, suffers from an important drawback: there is no way to evaluate the progress of the problem-solving process before reaching a desired or a sink state. As a result, often times, the problem-solver may pursue dead-end reasoning paths for a long time before its failure becomes evident.

> To be able to evaluate the progress of the problem solving, the monitoring task requires an abstract specification of what constitutes a successful problem-solving process, both in terms of what its final results should be, as well as, what its intermediate steps should be.

To accomplish the second aspect of its function, the monitoring task must produce a record of the problem-solving process that specifies all the design elements of the problem solver that were involved in that process. In the case of failure, this record will define the space of hypotheses that the subsequent blame-assignment task will explore to identify the cause of the failure. Essentially, the blame-assignment task will localize the cause of the failure in the malfunction of one of the elements involved in the failed problem-solving process.

> To provide a comprehensive hypothesis space for the blame-assignment task, the record produced by the monitoring task must be expressed in a language that can capture all the different types of the problem-solver's design elements that play a role in its reasoning, and that can potentially be at fault.

To fulfill the first requirement of the monitoring task, the model of the problem-solving process must specify the functional semantics of the overall problem solver, as well as the functional semantics of all its design elements that give rise to its reasoning. By *functional semantics* of the problem solver, we mean a specification of the nature of the problem-solving task that it is designed to accomplish. By functional semantics of a design element of the problem solver, we mean a specification of the role that this element plays in the accomplishment of this task. To fulfill the second requirement, the vocabulary, i.e., the ontology on which this model is based, should cover the range of the design elements that give rise to the problem-solver's reasoning behavior.

2.2 Blame Assignment

The blame-assignment task takes as input a description of the symptoms of the problem-solver's failure, potentially some feedback from the external environment regarding the behavior desired of it, and the trace of the failed problem-solving process. It has the goal of producing as output as set of potential causes of this failure [Minsky 1963, Samuel 1959].

There are several aspects to the problem of blame assignment that make it especially hard:

1. the blame-assignment task has to set up the information context for the subsequent repair task, that is, it has to postulate *what needs to be learned*,

2. there are several different types of causes for a problem-solver's failures,
3. there can be several causes for the problem-solver's failure to solve a particular problem, and
4. the localization of the cause(s) for the failure in a particular problem-solving episode is hard, especially for complex problem solvers.

The role of the blame-assignment task in the context of failure-driven learning goes beyond postulating hypotheses for cause(s) that can potentially explain why the failure occurred, a role that that the blame-assignment-as-abduction view might suggest. Upon the detection of a failure, the blame-assignment task needs to decide *what needs to be learned* from the failure under investigation. That is, it has to suggest ways in which the problem solver can potentially be modified so that similar failures can be avoided in the future. The class of self-adaptations that the system is able to accomplish is equivalent to the class of learning tasks for which the blame-assignment task is able to postulate a need. Therefore, it is important to note that the taxonomy of the learning tasks recognizable by the blame-assignment task should be complete with respect to the types of performance improvement that the system needs to be able to accomplish.

> The need of the blame-assignment task to specify *what to learn* gives rise to the need for a complete taxonomy of learning tasks, specified at a level operational for the repair task.

In complex, multi-strategy problem solvers it is unrealistic to assume that all the causes of all the problem-solver's failures are of the same type. Thus the blame-assignment task has to admit a taxonomy of causes of failures. The degree to which this taxonomy is complete and the degree to which each of the types of causes can be detected defines the effectiveness of the blame-assignment task to discern the causes for the problem-solver's failures. Consequently it defines what the overall learning process will be able to learn. At a very high level of abstraction, there are two types of causes that can potentially lead the problem-solving process to fail:

1. some part of the problem-solver's domain knowledge may be incorrect or incomplete, or incorrectly organized and therefore inaccessible, or,
2. the design elements that give rise to its problem solving may be incomplete, or incorrect or inappropriately organized, and thus, although the problem solver has sufficient domain knowledge, it cannot use it appropriately.

Not many failure-driven systems admit the possibility that either one of these types of errors may have caused the failure of the problem solver. Traditionally, failure-driven learning methods have assumed either that the system's domain knowledge is perfect, and therefore the cause of the failure lies in the problem-solving process [Carbonell *et al.* 1989, Hammond 1989, Laird *et al.* 1986, Mitchell *et al.* 1981, Samuel 1959, Sussman 1975] or that the problem-solving process is perfect, and therefore the cause of the failure must lie in its domain knowledge [Davis 1977, Doyle 1979, Simmons 1988].

> To be able to recognize errors in its domain knowledge, the blame-assignment task needs to have an explicit understanding of the types of domain knowledge that are generally used by the problem-solving process.

Otherwise, when the problem-solving process fails to draw an appropriate inference, the blame-assignment process could not assign blame to errors in that piece of knowledge on which this inference would be based.

If the problem solver has all the domain knowledge relevant to successfully solving a particular problem and still fails, then, in principle, there are two possible reasons for that failure:

1. either the solution is completely beyond the class of solutions the problem solver is capable of producing, or

2. the problem solver can potentially produce multiple solutions for any given problem, and the desired solution was overseen in favor of another undesirable one.

It is important to note here that these types of causes are especially difficult to detect; in fact, they are impossible to detect based solely on the trace of the problem-solving episode. The trace of the problem-solver's reasoning on a particular problem describes simply a single instance of the problem-solving behavior that the system is able to exhibit. It does not provide any information regarding the overall range of this behavior. Thus, if the only information available is a failed reasoning trace, the blame-assignment process does not have any basis for inferring whether the problem solver could have solved the problem or not.

> To be able to infer whether or not the problem-solver's design elements could have produced the desired solution, the blame-assignment task needs an abstract characterization of the classes of inferences that these elements are intended to produce.

Finally, it is important to consider the complexity of the blame-assignment task especially in the context of sophisticated, multi-strategy problem solvers. Even though the trace of the failed problem-solving episode bounds to some degree the space of the possible causes of the failure, in the case of complicated problem solvers, this space is still extremely large.

> In order for the blame-assignment process to be efficient, it is necessary to explicitly specify the composition of the problem-solver's higher-level design elements in terms of the lower-level ones (preferably in a hierarchical manner) so that the blame-assignment process can efficiently acquit large subsystems of the problem solver and focus fast on the failing ones.

To fulfill the requirements of the blame-assignment task, the model of the problem solver must specify the types of problem-solving tasks accomplished by the problem solver. That is, it must specify the problems it can solve and the nature of the solutions it produces for these problems. Furthermore, it must specify the types of domain knowledge that are used for each type of inference made during problem solving, and how these inferences are composed to produce the overall solution of the problem solving process. Finally, the vocabulary, on which the model is based, should give rise to a taxonomy of learning tasks, "complete" with respect to the types of performance improvement that it needs, and operational with respect to the methods used by the subsequent repair task.

2.3 Repair

Given the set of potential causes for a given problem-solving failure, as identified by the blame-assignment task, and the feedback regarding the behavior desired by the failing problem solver, the repair task has as a goal to adapt the failing problem solver so that the cause(s) of its failure are eliminated. To that end, the repair task has

1. to decide on which cause(s) to address, and

2. to modify the problem solver in a way that the cause of the failure is eliminated and the overall consistency of the problem solving is maintained.

The complexity of the repair task in a failure-driven learning process depends on whether this process assumes a single cause for each problem-solving failure, and on whether it assumes a single method for each different type of learning task it can accomplish.

> If neither the single-fault nor the single-strategy-per-type-of-cause assumptions are made, then, the repair task requires a set of criteria for selecting one among the possible causes and a second set of criteria for selecting one among the methods possibly applicable to it.

Once a particular learning task has been selected to be accomplished, and a particular method has been chosen, the goal becomes to actually perform the selected modification and to propagate its consequences so that the result is a consistent problem-solving process. The specific process of instantiating a particular modification depends on the type of the modification. If for example, a modification is performed to the domain knowledge of the problem solver, the constraints that relate the modified piece of knowledge with other types of domain knowledge must be propagated so that the overall domain knowledge is consistent. If, on the other hand one of the problem-solver's design elements is modified, its inter-dependencies with the other existing design elements must be taken into account so their overall composition remains consistent.

> To enable the consistent repair of the problem solver, the system needs an explicit specification of the inter-dependencies among the different types of knowledge and the inter-dependencies among the different design elements of the problem solver.

To meet the knowledge needs of the repair task, the model of the problem solver should give rise to two sets of criteria. The first set should enable the learning process to select among the learning tasks possibly appropriate in the context of a failed problem-solving episode. The second one should enable it to select among the learning methods potentially applicable to the chosen task. Furthermore, the model should specify how the problem-solver's design elements interact in order to accomplish its task, i.e., the problem-solver's *compositional semantics*, and the inter-dependencies among the different types of knowledge available to it.

2.4 Verification

Once the repair task has been completed, the goal of the learning process becomes to evaluate whether it was successful or not. The verification task takes as input the specification of the problem that led to failure in the most recent problem-solving episode, the modified problem solver, and the feedback. It has as a goal to evaluate whether the repair was successful.

To that end, the verification task may evaluate the modified problem-solver's behavior on the same problem that led to failure before, against the behavior desired of it on that problem. If they are the same, the repair can be considered successful. It is important to note here that a repair task which makes informed decisions on how to consistently modify the problem solver is much more likely (although not guaranteed) to be successful than another repair task which arbitrarily modifies the problem solver. If the new behavior is again different from the desired one, the learning process may try to perform an alternative modification which was suggested but not performed in the last learning episode, or it may attempt to learn from its new failure.

2.5 Summary

The overall process of reflective learning consists of four subtasks: monitoring, blame assignment, repair, and verification. The role of the monitoring task is to evaluate the progress of the problem-solving process, to recognize when it fails, and to record the design elements involved in the

process so that the subsequent blame-assignment task can evaluate their behavior. The role of the blame-assignment task is to identify the cause of the problem-solver's failure. The role of the repair task is to select one of the possible causes identified by the blame- assignment task, and appropriately modify the problem solver in order to eliminate it. The role of the verification task is to evaluate whether the performed modification was indeed successful in eliminating the causes of the problem-solver's failure. To accomplish these subtasks the system needs a specification of the expected correct behavior of the overall problem solver and of its design elements, i.e., their functional semantics. In addition, it needs a specification of how these elements interact to accomplish its overall tasks, i.e., their compositional semantics. Finally it needs to organize this knowledge in a way that enables its efficient access. The different subtasks of this reflective process, and the methods developed to accomplish them are discussed in detail and illustrated with examples from AUTOGNOSTIC in subsequent chapters.

CHAPTER III

A CONTENT THEORY OF PROBLEM SOLVING

Two questions arise in any effort to model an artifact, the answers to which have a critical impact to the usefulness of the model:

1. What should be the content of the model, i.e., what are the aspects of the artifact that need to be specified in the model?
2. What should be the representation and organization of that content, so that it can be effectively used by the process that this model is intended to support?

In general, the answer to the first question should be decided based on the task that the model is intended to support. That is, the model should capture this knowledge about the modeled artifact which is useful for drawing the inferences needed for the task at hand. The purpose of this work is to develop a self-redesign process, which can identify the shortcomings of a problem solver both in its domain knowledge and in its problem-solving process. Given the knowledge requirements of the subtasks of this process, as analyzed in the previous chapter, the model of the problem solver should describe its task structure, in terms of

- its tasks, i.e., the functional semantics of the design elements of which it is composed,
- its methods, i.e., the compositional semantics of the interactions of these design elements, and
- its knowledge on which the functioning of these design elements relies.

The answer to the second question is, in general, motivated by the need for the process, accomplishing the task in question, to be efficient. An appropriate representation scheme is one that organizes together all the related aspects of the model, so that the reasoning process has only to consult "localities" of information at any step. The greater the extent to which reasoning becomes local, the less search is needed, the greater the efficiency of the process. Based on the efficiency requirements of the blame-assignment task, as analyzed in the previous chapter, the model of the problem solver should

- organize the knowledge about the composition of the problem-solver's design elements in a hierarchy,
- where higher-level design elements index the lower-level elements they are composed of.

The rest of this chapter discusses these two issues, outlines and compares the approaches explored by earlier AI research, analyzes the specific answers adopted in this thesis, and illustrates these answers with the model of one of the problem solvers AUTOGNOSTIC has been integrated with.

3.1 Content of the Model: Task Structures

Newell [1982] proposed the *knowledge level* as the appropriate level of description of intelligent systems. At the knowledge level only the content of the system's knowledge and the goals this knowledge serves are specified. At this level, there is no description of the structures that hold the system's knowledge. Furthermore, there is no completely specified processing mechanism but rather a general principle characteristic of a set of possible alternative processing mechanisms. Finally, there is only a very abstract and non-deterministic description of the system's behavior. Marr [1982] too identified three different levels at which it is possible to formulate theories about complex information-processing systems. At the highest level, one can simply describe its mapping from its input to its output information. At the next level, one can describe the representation of its input and output, and also the nature of the algorithm that carries out the transformation. At the third and most detailed level, one can describe how the algorithm and the representation are implemented on the medium that carries out the transformation, i.e., the computer architecture.

In a similar vein, Chandrasekaran developed task-structures as a framework for analyzing expert problem solving [Chandrasekaran 1987, Chandrasekaran 1989, Chandrasekaran and Johnson 1993]. The idea of task-structure analysis can be traced back to the idea of *generic tasks* [Chandrasekaran 1983]. Generic tasks are types of elementary inferences, instances of which were found in a wide range of complex information-processing tasks in different domains. All instances of a generic task can be accomplished by the same method and require similar types of knowledge. Thus, generic tasks can be viewed as "building blocks" for intelligent problem-solving behavior. As the idea of generic tasks was applied to different tasks and domains, the need for flexibility in task modeling became apparent. Task modeling should allow the specification of multiple types of knowledge available in a domain, potentially supporting multiple methods for accomplishing the same task. These needs gave rise to the task-structure analysis. In task-structure analysis, the emphasis is not in identifying instances of a pre-defined set of primitive blocks in the reasoning process, but rather in analyzing the elementary inferences that occur in this process and their interactions. Some of Clancey's [1985, 1986] , McDermott's [1988] , Steels' [1990] , and Wielinga and colleagues work [1992] also shares this functional perspective.

I have adopted task structures as a content theory for the model of the problem solver. This theory specifies the tasks that the problem solver accomplishes, the methods it uses to do that, and the knowledge that is necessary for the application of these methods. A task is specified by the types of information it consumes as input, the types of information it produces as output, and the nature of the transformation it performs between the two. It can be accomplished by one or more methods, each of which may decompose it into a set of simpler subtasks. A method is characterized by the kinds of knowledge it uses, the subtasks it sets up, and the control it exercises over the processing of these subtasks. These subtasks can, in turn, be accomplished by other methods, or, if the appropriate knowledge is available, they may be solved directly. The task structure of a problem solver thus provides a recursive decomposition of its overall task in terms of methods and subtasks. To provide a basis for comparison with the other widely used view of problem solving, i.e., problem solving as search in a problem state space, tasks can be thought of as roughly equivalent to goals. "Leaf" tasks in particular, that is, tasks not further decomposable by methods,) can be thought of as elementary goals which can be immediately accomplished by operators. Methods do not really have an equivalent in that view, but they can be thought of as general plans or strategies for how the solutions to different subgoals combine to accomplish higher-level goals.

Notice that the description of a problem-solver's task structure captures the types of knowledge needed by the learning subtasks as they were analyzed in the previous chapter. For example, the tasks, the methods and the knowledge of the problem solver are the elements that can potentially be at fault and cause the problem solver to fail. By specifying these elements, the model captures the complete set of elements that blame assignment might need to inspect. Further, notice that the description of the nature of the information transformation accomplished by the overall task of the problem solver defines the class of problems which the problem solver is able to solve. In addition, the specification of the information transformation accomplished by each subtask of the problem solver constitutes an abstract, problem-independent specification of correctness for all the inferences that this task might contribute to a specific problem-solving episode. The recursive decomposition

of the task into a task structure organizes the problem-solver's design elements in a hierarchy, in order to support an efficient blame-assignment process. Finally, the description of the information and control interactions among the problem-solver's subtasks capture the inter-dependencies among the problem-solver's design elements, in order to enable its consistent repair.

3.1.1 An Example: Router's task structure

ROUTER [Goel and Callantine 1992, Goel *et al.* 1993, Goel *et al.* 1995], is a path planning system. Its task is, given an initial and a goal location in a physical space, to find a path between them. ROUTER has two different types of knowledge about its domain: a topographical model of its world, and a memory of previously planned paths in this world. The model is organized hierarchically, at several different levels of spatial abstraction. Each spatial abstraction corresponds to a neighborhood that contains knowledge about some area in ROUTER's world. High-level neighborhoods describe large spaces in little detail. They get decomposed into lower-level neighborhoods, which cover a subspace of their parent neighborhood, in greater detail. ROUTER's memory is also organized around the neighborhoods of its world model: each path in the memory is indexed in terms of its initial and final locations and the neighborhoods in which these locations belong. ROUTER knows two different methods that it uses to solve the problems it is presented with: a model-based method and a case-based one. The former makes use of its world model, where the latter one makes use of its path memory. Each one of these methods is associated with a sponsor. The sponsor of a method is responsible for evaluating the applicability and the utility of the method in question, for each particular problem presented to ROUTER.

ROUTER was designed and developed independently of this work; it does not have a model of its own reasoning, nor does it reflect on it. For the purposes of this work, the task-structure model of ROUTER's reasoning was developed, a diagrammatic representation of which is presented in Figure 3.1. For a detailed description of ROUTER's SBF model see Appendix 1. In this figure, the tasks are shown as single-line parallelograms, where the methods are shown as double-line rectangles.

ROUTER's overall task, route-planning, is, given an initial intersection, a final intersection, and the general space in which the problem should be solved, top-neighborhood, to produce a path between the two.

The route-planning task is accomplished by the route-planning method, which decomposes it into a sequence of four simpler subtasks, classification, path retrieval, search and storage. For the classification subtask, ROUTER uses its world model to identify the neighborhoods in which the initial and final locations belong. The output of the classification subtask consists of an initial and a final zone. The semantics of the classification task specify the nature of the information transformation it performs, that is, that the initial intersection must belong in the initial zone, and the destination intersection must belong in the destination zone.

The second subtask is path retrieval. ROUTER probes its path memory with the two problem locations and their respective neighborhoods in order to retrieve a similar, previously solved, problem. If the case-based method is used to solve the problem, the path retrieved from memory will be later used as a subpath of the overall desired-path, i.e., the middle-path. Furthermore, the retrieved path sets up two intermediate intersections, int1 and int2. If the problem is solved by the case-based method, these two intersections will constitute the ending and beginning points of the other two subpaths of the overall desired-path, i.e., first-subpath and third-subpath respectively.

The next subtask is the actual search task, for which ROUTER knows two different methods: the model-based method, and the case-based method. The sponsor of the former method suggests that it is always applicable, where the sponsor of the latter one suggests that it is applicable only when an appropriate path has been retrieved from memory. For the sake of simplicity, in the context of AUTOGNOSTICONROUTER, the case-based method's sponsor was modified to require that the two problem locations do not belong in the same neighborhood in order to propose that method. In cases where both methods are applicable, ROUTER selects one arbitrarily. Once the selected method has been completed and it has produced a path, ROUTER stores it in its memory for reuse in the future.

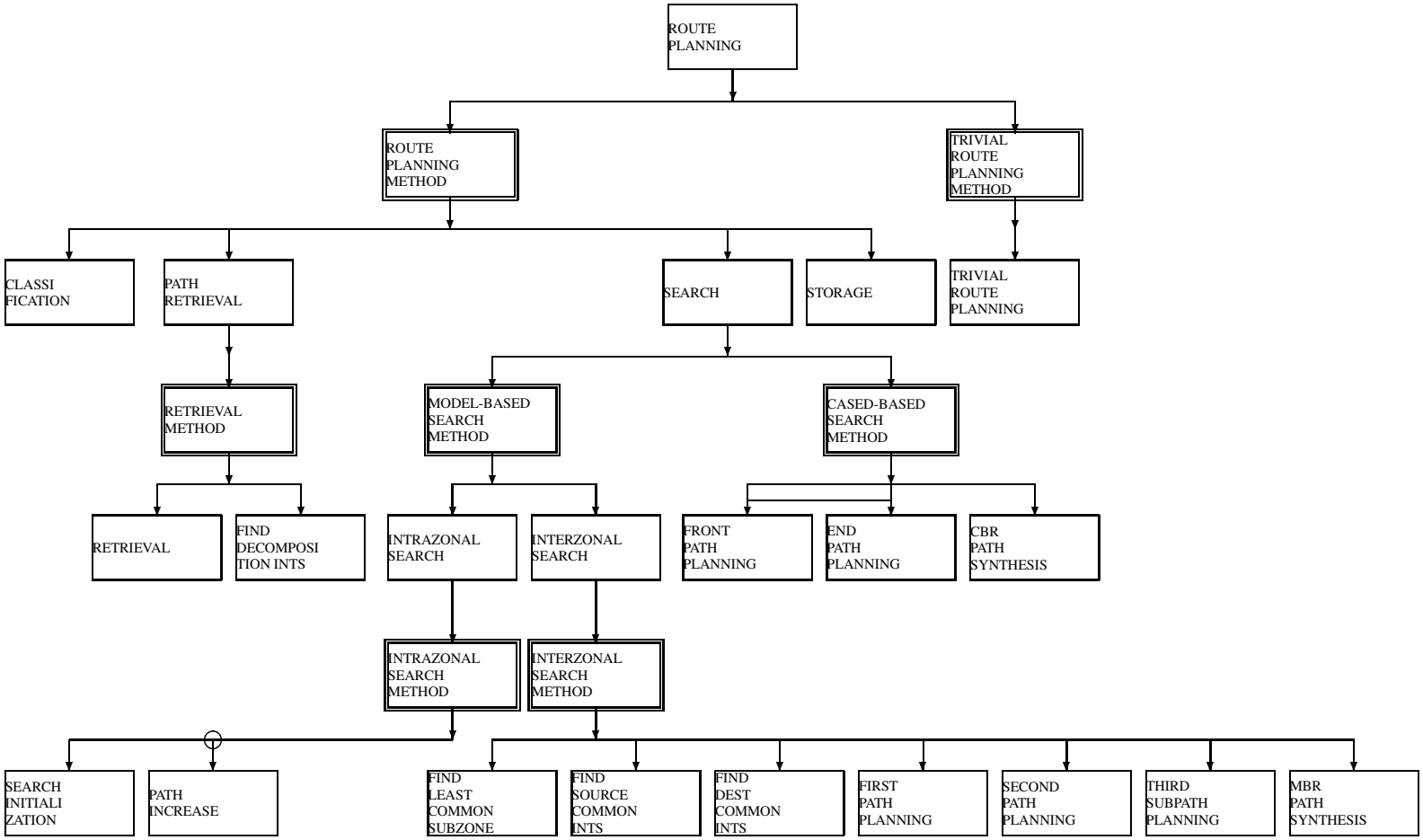


Figure 3.1: ROUTER's task structure.

Each method decomposes the search task into a different set of subtasks. For example, the model-based method sets up the subtasks of intrazonal-search , and interzonal-search . The first subtask is only applicable when the two locations belong in the same neighborhood, where the second one is applicable when the two locations belong in different neighborhoods. The intrazonal-search subtask is accomplished by a breadth-first-search method which sets up two subtasks: the search-initialization subtask, and the path-increase subtask. The former sets as first node of the path the initial location, and the latter adds recursively nodes to this path, until the final location is reached. The small circle before the path-increase subtask in Figure 3.1 denotes the control operator $loop_{until}$; essentially, it specifies that this subtask is performed repeatedly until a condition is met. Finally, Figure 3.2 depicts the task structure of the step-in-increase-path task, which is the prototype for each repetition of the path-increase task.

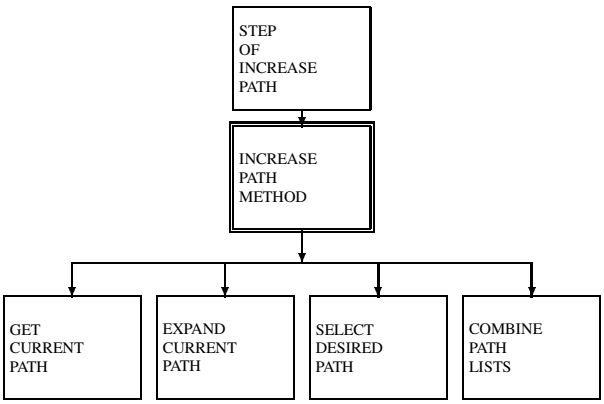


Figure 3.2: The task-structure decomposition of step-in-increase-path .

The interzonal-search task is decomposed into the following subtasks: find-least-common-subzone , find-source common-int , find-dest-common-int , middle-subpath planning, first-subpath planning, third-subpath planning, and mbr path-synthesis . Its first subtask is find-least-common-subzone . This task has as a goal to identify a neighborhood, usually a higher-level one than either the initial or the final one, which covers a space that includes the spaces covered by both the neighborhoods in which the initial and final locations belong. Next are the subtasks of finding a source-common-int and dest-common-int which produce two intermediate locations int1 and int2 correspondingly, that are used as intermediate nodes in the desired path. The double-pointed arrow connecting these subtasks denotes that these two subtasks can be performed *in_parallel*, that is, in any order relative to each other. Furthermore, both these subtasks perform essentially the same information transformation, although with different input information, and in service of different subgoals of problem solving. They, both, produce an intersection common to the two neighborhoods they are given as input. For that reason, these two subtasks are recognized as instances of a common prototype task, find-common-int .

The selection of these two locations results in the decomposition of the overall problem into three simpler subproblems: the planning of a path from the initial-location to int1 , the planning of a path from the int1 to int2 , and, finally, the planning of a path from the int2 to the final-location . These three subtasks are the first-subpath planning subtask, the middle-subpath planning subtask, and the third-subpath planning subtask, and

they can be performed in any order relative to each other. The first and third of them are instances of the planning task, while the second one is an instance of the intrazonal-search task. This is because, the nature of the information transformations that produce the intersections int1 and int2 ensure that these two intersections belong in the same neighborhood. The last subtask set up by this method, mbr path-synthesis, is the synthesis of these three paths into a single path which constitutes the solution to the overall problem.

Finally, the case-based method decomposes the planning task into yet another set of subtasks. Its subtasks are similar to the subtasks of the model-based method when the initial and final locations belong in different neighborhoods. This method also decomposes the overall problem into three simpler ones, one of which is solved by the retrieval of a previously produced path from memory. The other two subproblems are, first planning a path from the initial-location to the first node of the middle-path, the front-path planning subtask, and from the last node of this path to the final-location, the end-path planning subtask. Finally, as in the previous method, the last subtask cbr path-synthesis combines the three paths into the solution of the overall problem. Both the mbr path-synthesis and the cbr path-synthesis tasks are instances of the same task, recomposition.

3.2 A Taxonomy of Learning Tasks: Modifications to Problem Solving

The adoption of a particular view towards problem solving gives rise in a taxonomy of modifications that the learning process may be able to perform to a problem solver. For example, if problem solving is viewed as a search in a problem space, then, learning can potentially introduce new operators to a problem space, or modify the existing ones, or control the selection process among them when there are several of them applicable. Consequently, the content captured in the model of the problem-solving process defines the types of modifications that the learning process will be able to perform on a particular problem solver. As a result, it decides the improvements to the problem-solving performance that learning will be able to produce.

The task-structure view of problem solving gives rise to two types of modifications to the problem solver:

1. modifications to the content and the organization of the problem-solver's domain knowledge, and

2. modifications to the content and organization of the design elements in its functional architecture, i.e., its tasks and methods.

Intuitively, these modifications seem to be complete with respect to the three performance-improvement goals that learning must fulfill. The section below discusses several scenarios of problem solvers whose performance would benefit from each one of these types of modifications.

3.2.1 Modifying a task in the task structure

Modification of a task of the problem solver constitutes modification of one of the design elements of its functional architecture, that is, modifications of the content of its functional architecture. Given that a task is defined in terms of the types of information it takes as input and produces as output, and the nature of the transformation between the two, modifying a task in the task structure may involve the modification of any of these three elements.

1. **Modifying the input of a task**

   For example, consider the situation where an agent knows how to plan paths from one intersection to another. Let us also assume that this agent is asked to plan a path from a landmark, let's say the Tech Tower, to an intersection. Clearly, the essentials of path planning remain the same. However in order for the agent to be able to solve this new problem, it has to expand the class of input it accepts, from a pair of intersections to a pair of locations, where

a location can be either an intersection or a landmark. The path-planning process itself may also have to be modified after this extension to accommodate this extended class of input. Or, if the current path- planning process is general enough, such a modification may not be necessary. Given that the modified planner accepts a wider class of input locations which it can connect, it solves an extended class of problems than the original one.

2. **Modifying the output of a task**

   Consider for example the agent described above. Let us assume that this agent is asked to produce, in addition to the path between the two given locations, an estimation of the time it will take to traverse this path. This information could be useful to another agent who might be scheduling errands. In this case, our agent needs to expand its "notion" of what constitutes the output of its path planning, in order to include in this notion, an attribute of the path, namely "the cost of the execution of the path". In this case again, the modified agent solves an extended class of problems than the original one, in the sense that for each problem it produces the solution that the original agent would have produced, and then some more information.

3. **Modifying the transformation performed by a task**

   Clearly, in both the above examples, where the types of the task's input or output change, it may become necessary to also modify the relations that define the transformation that the task carries out between them. However, even if the input and output remain the same, it may still be beneficial for the agent to modify the relations that partially define this transformation.

   For example, let us assume that our agent specifies as "path planning", the production of a path such that, its initial node will be the input source location, and its final node will be its input destination location. Now, if our agent is interested in doing path planning for mail delivery in a small area, then it could be beneficial to also specify that the length of the paths it produces should be less than a maximum limit, say "4 nodes long". This could be a valid definition of a path, if the area in which the agent delivers mail is so connected that all pairs of location are "closer than 4 intersections long". Such specification would help the agent identify whether a given problem is outside its scope of mail delivery, namely when there is no path "smaller than 4 intersections long" then the destination is outside the agent's scope.

3.2.2 Modifying the decomposition of a task by a method

Given that a method is defined by the set of subtasks it sets up for the task it decomposes, and the control it imposes over these subtasks, the possible method modifications are modifications to the set of its subtasks or to the control among them. Modification of a method of the problem solver constitutes modification to its functional architecture. If a new task is added to (or deleted from) the set of the problem-solver's tasks, then the set of the problem-solver's design elements is modified, that is the content of its functional architecture is modified. If the control of processing among its tasks is modified, then the functional architecture is simply reorganized.

1. **Modifying the set of subtasks**

   (a) **Adding a task in the set**

   Take for example the case where the agent knows how to plan paths between intersections but does not know how to plan paths between landmarks, as described in the first scenario. One possible way for the agent to expand its path-planning capabilities, in order to able to handle landmarks, in addition to intersections, is to add a pre-processing task in the set of tasks that the path-planning task gets decomposed into. This task can take as input the input locations, and if they are landmarks it can return as output the intersections closest to them. If they are intersections, it can return them without any transformation. The rest of the path-planning subtasks can continue as before. This modification assumes, of course, that the agent already knows that landmarks are objects

located on streets, between two intersections. Thus, the agent will be able, given two landmarks to return a path connecting the two intersections next to which they lie.

(b) **Deleting a task from the set**

Let us assume for example that the agent has been planning paths in a domain where there is a very highly interconnected set of high-speed highways. As a result, the agent has developed the "habit" of "searching for major highways to use" except for the initial and final steps of entering and exiting a highway. The way this habit is instantiated in this agent's task structure, is by a task which filters out the non-highway pathways unless when looking for entry and exit points to the highway system. Let us imagine, that this agent moves to an environment where such a highway network is not available. In such a case, the habit of using only highways for planning paths can be overly constraining. As a result the agent may start failing to produce paths. Such failures should help the agent realize that its habit to look for highways is imposing unnecessary, and even unsatisfiable in the new environment, constraints. Consequently the agent might remove this "filtering task" from its task-structure.

(c) **Aggregating two or more subtasks from the set into one**

This kind of task-structure modification is roughly equivalent to the creation of macro-operators. Generation of macro-operators and compilation of knowledge has been shown to be a plausible mechanism for the phenomenon of "skill acquisition". Agents become more proficient in their problem-solving skills by aggregating a sequence of "individually interpreted" steps into chunks of opaque, internally non-inspectable, mechanisms. Having a model of what the individual steps were meant for in the beginning and a history of how (under which assumptions) the chunks were formed, may be useful for redesigning these chunks or for recognizing more easily when and why these chunks should be broken into their constituent parts.

2. **Modifying the control of processing among the method subtasks**

The control that a method imposes over the subtasks into which it decomposes a task can be modified by reordering these subtasks. Often, the order in which tasks are accomplished, when it is not completed determined by their knowledge requirements, can be decided or affected by several context requirements, such as relation between problem solving and execution. Consider for example a path planner who first decides on a major highway to connect the neighborhoods of its current location with its destination, and then works out the details on how to get on and off that highway. If this planner was integrated in a car, then it would probably be desirable to interleave its planning with its execution as soon as possible. That is, it might prefer to avoid thinking for a long time about the highway in a static mode. In such a case it would be better to first plan the initial segment to its nearest highway, and then find its way through the highway system to its destination.

3.2.3 Modifying the knowledge

1. **Modifying the content of the knowledge**

One way to expand the class of problems that a problem solver can solve, without modifying the tasks it knows how to accomplish, is by expanding its world knowledge. Acquisition of new world knowledge, in general, leads to increase of the domains in which the problem solver knows how to accomplish its tasks. For example, if the path planner increases its model of the navigation space, i.e., if it develops a model of a larger navigation space than what it used to, then it can plan paths for a greater number of pairs of locations. That is, it can solve more problems than it used to.

2. **Modifying the organization of the knowledge**

The more knowledge intensive the problem-solving process becomes the more important the organization of the knowledge becomes. If the problem solving is exhaustive search, then it

will always produce a solution for any problem, if there is a solution for the problem and if the problem solver knows all its necessary elements. However, as the space of possible problems and possible solutions increases, exhaustive search becomes unacceptably inefficient and introduction of an intermediate space, between the problem and the solution spaces, becomes necessary. Consider for example the problem of simulating the behavior of a complex device. For the description of such a device a large number of differential equations may be required. Usually, not all of these equations are relevant at each point in the device functioning. Thus, in order for the simulator not to have to consider all of these equations at each clock cycle, these equations can be clustered around different stages in the device functioning where they may be applicable.

3. **Modifying the representation of the knowledge**

   There has been a lot of work [Amarel 1968, Simon 1972] on the affects of representation to problem solving performance. Inappropriate representation of the problem space may lead to inefficient problem-solving (i.e., missionary and cannibals problem). Moreover, certain problems may become unsolvable because of inappropriate representations (i.e., the mutilated checkerboard problem).

3.3 Organization and Representation of the Model: The SBF language

Once the content that the modeling framework should convey has been decided, a representation and organization scheme needs to be developed, which is expressive enough to capture the information conveyed by the model, and which provides the organization to make this information accessible to the tasks that need to use it.

The task-structure view of problem solving is a functional view of problem solving. Viewed as a designed artifact, a problem solver has as a function to carry out a range of high-level information-processing tasks. In turn, each one of these tasks is accomplished through the synthesis of simpler functions accomplished by the elements of the problem-solver's functional architecture. Similar functional theories have been developed for another class of designed artifacts, namely physical devices. Therefore, a language developed for functional modeling of physical devices would be a natural starting point for developing a language for modeling the task structures of problem solvers. The starting point used by this thesis was the structure-behavior-function (SBF) language for modeling physical devices.

3.3.1 SBF Models of Physical Devices

The SBF language for modeling how physical devices work [Goel 1989] describes three aspects of the device:

- the device function,
- the device structure, and
- the device internal behaviors.

The function of the device is defined as the purpose for which it was designed. A device may exhibit several external behaviors; its function is the subset of these behaviors that were intended by the designer. The structure of the device is defined as the set of physical elements that comprise the device and the relations between them. The ontology, most commonly used to describe the structure of physical devices, has been based on components and substances [Bylander and Chandrasekaran 1985, Hayes 1979]. The internal behaviors of the device are the causal processes that occur internally in the device while it functions. These causal processes result

from the interactions among the structural elements of the device and, in turn, they result in the delivery of the device function.

Several modeling frameworks for physical devices have been developed [Goel 1989, Rasmussen 1985] in which the same three aspects of a device are specified. In the particular language that was used as a basis for the modeling language of this thesis, the function of the device is specified as a transition from an input behavioral state to an output behavioral state. A behavioral state consists of a partial description of the device components and substances at some particular point in its functioning. The internal behaviors of the device are specified as hierarchically organized sequences of state transitions accomplished due to functions of the device components or causal interactions between them. For each one of the device functions an internal behavior is specified which explains how this function is accomplished through internal causal processes, and this internal behavior is indexed by its corresponding device function. Finally, the structure of the device is specified as a set of components, organized in a partonomic hierarchy, and related with each other and with the device substances by relations such as connectivity and containment.

In an analogous way, in the SBF language for modeling problem solvers[1], a task is specified as a transition from an input information state to an output information state, where an information state is a partial description of the information available to the problem solver at some particular point in its reasoning. The task is also annotated by a set of semantic relations between its input and output information state, a pointer to the prototypical task of which it is an instance, and pointers to the methods than can potentially accomplish it, or alternatively, a pointer to the procedure which accomplishes it. The semantic relations of the task constitute a partial description of the expected, correct performance of the task. The internal behaviors of the problem solver are specified as sequences of information transformations that accomplish the overall tasks of the problem solver. Each sequence corresponds to a method applicable to a task, and each one of the information transformations in the sequence corresponds to a subtask resulting from the decomposition of that task by that method. The subtasks that are accomplished through further decomposition by problem-solving methods act as indices to the information-transformation sequences corresponding to these methods. The subtasks which are directly accomplished through the application of some problem-solving procedure, i.e., the leaf subtasks, act as indices to the procedures which accomplish them.

In an extension of the task-structure framework, the SBF model of a problem solver also includes the description of the domain knowledge that the problem solver uses in its reasoning. The domain knowledge is specified in terms of the types of domain concepts that the problem solver knows about and the relations applicable to them. The types of information that the problem solver manipulates while reasoning are instances of these types of domain concepts, and the semantic relations of its tasks are often expressed in terms of domain relations.

The set of the procedures which accomplish the leaf subtasks of the problem-solver's task structure and the data structures holding the problem-solver's domain knowledge constitute its structural elements.

3.3.2 The Grammar of SBF language for Problem Solving

3.3.2.1 The Task

Each task is described as a tuple $T := (i, o, \{p\}, by/op, c, sub, s)$, where i is the input of the task, o is its output, p is another task of which the current task is an instance, by is a set of methods which can be applied to accomplish the task, op is a procedure which can be invoked to accomplish the task (if it is a leaf task), c is a set of conditions under which the task should be performed, sub is the task at the immediately higher-level in service of which the task at hand is spawned, and s is a set of semantic relations partially describing the role of the task in the task structure, i.e., the nature of the transformation it performs between its input and output.

i and o are information states. For some specific task t_i, the information state $i(t_i)$ consists of the types of information which are necessary for the task t_i to be accomplished. Thus, for each

[1] Henceforth, I will call it simply the SBF language.

type of information, $i \in i(t_i)$, i is either required by some of the subtasks resulting from the decomposition of t_i by some method $m_i \in by(t_i)$, or it is an input parameter of the procedure which implements $op(t_i)$. For the same task t_i, the information state $o(t_i)$ consists of the types of information which are produced by the task t_i and which are going to be used by other subsequent tasks.

$c(t_i)$ and $s(t_i)$ are sets of relations. The relations in $c(t_i)$ describe the conditions under which the task t_i contributes to the progress of the reasoning. They are used as a criterion to decide whether or not to perform the task in the context of a particular problem-solving episode; thus they refer to types of information available in $i(t_i)$. The relations in $s(t_i)$ partially describe the information transformation that t_i performs in the service of the overall reasoning. Essentially, they describe the function intended of the task in the overall context of reasoning. Thus, each relation in $s(t_i)$ is either a unary relation on some type of information in $o(t_i)$ or it relates some type of information in $o(t_i)$ with some type of information in $i(t_i)$.

$by(t_i)$ is a set of methods which are applicable to the task t_i. When some of these methods is applied to t_i, it sets up a set of subtasks which if accomplished, the overall information transformation intended by t_i will be accomplished. Hencefort, the symbol $tree(t_i, m) : m \in by(t_i)$ will be used to denote the task structures which may possibly result from the application of m to t_i. The attribute by establishes a partonomic hierarchy among tasks.

Finally, $p(t_i)$ is another task, $t_{prototype}$, of which t_i is an instance. $t_{prototype}$ need not be a *generic task* in the sense that Chandrasekaran defined generic tasks in [Chandrasekaran 1983, Chandrasekaran 1987]. It is simply a task which accomplishes an information transformation more general than, or equivalent to, the transformation accomplished by t_i. If the task t_i does not have associated with it a method or a procedure, i.e., $by(t_i) = \emptyset \wedge op(t_i) = \emptyset$, then it can be accomplished with the methods or the procedures associated with $t_{prototype} = p(t_i)$. The attribute p establishes a taxonomic hierarchy among tasks.

In AUTOGNOSTIC the SBF language is implemented in a schema-based framework. Figure 3.3 depicts the schema for specifying a task, and the particular schema specifying ROUTER's overall task, i.e., path planning .

3.3.2.2 The Method

Each method is described as a tuple $M := (t_{overall}, c, t_{sub}, ctrl)$, where $t_{overall}$ is the task to which the method is applied, i.e., the task which the method is intended to accomplish, c is a set of conditions under which the method is applicable, t_{sub} is the set of subtasks that the method sets up for the $t_{overall}$, and $ctrl$ is a set of control relations over these subtasks.

For any specific method, m_i, $c(m_i)$ is a set of relations describing the conditions under which the method m_i is applicable to the task $t_{overall}$. They are used as a criterion to decide whether or not to apply the method in question to the task; thus they refer to types of information available in the information state $i(t_{overall})$.

When a method, m_i, is applied to the task $t_{overall}(m_i)$ it decomposes it into a set of simpler subtasks, $t_{sub}(m_i)$. The order in which these subtasks must be accomplished is not completely specified by the method. Instead, the method m_i imposes only a partial ordering among them which is specified in the $ctrl$ element of the *m-tuple*. In the SBF language, there are three control operators $serially$, $in\_parallel$ and $loop_{until}$. The control that the method imposes over the subtasks it sets up, $ctrl(m_i)$, is specified as a control operator imposed over a set of subtasks in the set $t_{sub}(m_i)$ or other control statements. That is, $ctrl := (control\_op, \{CT\}^*), CT := T, CT := ctrl$. When two subtasks are connected by the $serially$ operator they must be performed in the order they are mentioned. When they are connected by the $in\_parallel$ operator they may be performed in any ordering. Finally, when they are connected by the $loop_{until}$ operator then they must be performed repeatedly until the condition annotating the $loop_{until}$ operator is met.

Figure 3.4 depicts the schema for specifying a method, and the particular schema specifying ROUTER's intrazonal-search method.

task:	the name of the task, t
input:	the types of information it takes as input, $i(t)$
output:	the types of information it produces as input, $o(t)$
semantics:	the specification of the nature of the information transformation of the task, $s(t)$
methods:	the set of methods possibly applicable to the task, $by(t)$
procedure:	the procedure that directly accomplishes the task, $op(t)$
prototype:	the prototype task of which the task at hand is an instance, $p(t)$
conditions:	the conditions under which the task is useful, $c(t)$
subtask_of:	the super-ordinate task in service of which t is performed

task:	route-planning
input:	(initial-point final-point top-neighborhood)
output:	(desired-path)
semantics:	((same-point initial-node(desired-path) initial-point) (same-point final-node(desired-path) final-point))
methods:	(route-planning-method)
conditions:	(not(same-point initial-point final-point))

Figure 3.3: The task schema, and the specification of the route-planning task in the SBF language.

method:	the name of the method, m
applied to:	the task to which it is applicable, $t_{overall}(m)$
conditions:	the conditions under which the method is applicable, $c(m)$
subtasks:	the set of subtasks that the method sets up for the task it accomplishes, $t_{sub}(m)$
control:	the flow of control among these subtasks, $ctrl(m)$

method:	intrazonal-search-method
applied to:	intrazonal-search
conditions:	(zone-intersections(initial-zone final-intersection)
subtasks:	{ search-initialization path-increase }
control:	serial(search-initialization loop(path-increase))

Figure 3.4: The method schema, and the specification of the intrazonal-search method in the SBF language.

3.3.2.3 The Type of Information

The input and the output of each task in the problem-solver's task structure consists of a set of types of information. A type of information is defined as a tuple $I := (wo, t_{in}, t_{out})$.

For any specific type of information, t_{in} and t_{out} are, correspondingly, the sets of tasks which

consume and produce that type of information. That is, $\forall t_j \in t_{in}(i), i \in i(t_j)$ and $\forall t_j \in t_{out}(i), i \in o(t_j)$. Finally, the element wo in the *i-tuple* specifies the type of domain concept of which this information type is an instance.

Figure 3.5 depicts the schema for specifying a type of information reasoned about by the problem-solver's task structure, and the particular schema specifying the information tmp-path used in the step-in-increase-path task structure of ROUTER.

information:	the name of the information, i
type of object:	the type of domain concept of which it is an instance, $wo(i)$
input to:	the tasks that consume it as input, $t_{in}(i)$
produced by:	the tasks that produce it as output, $t_{out}(i)$

information:	tmp-path
type of object:	path
input to:	expand-current-path
produced by:	get-current-path

Figure 3.5: The type-of-information schema, and the specification of the tmp-path in the SBF language.

3.3.2.4 The Domain Concept

Domain concepts[2] are the concepts which are relevant to reasoning in a particular domain. They may correspond to physical entities in the real world or they may be abstract. A domain concept is described as a tuple $Cnpt := (d, attrs, id)$ where d is the domain in which this concept may take values, $attrs$ is a set of attributes characteristic of the concept, and id is a predicate that evaluates the identity between two instances of that concept.

The actual knowledge of the problem solver about the domain objects of this concept type is in the form of a set of instances that the problem solver knows about. This set of instances constitutes the domain d of this object type.

Finally, for each attribute, $attr$, of a concept, the SBF language specifies its $name$, its $type$, and a function f which, given a specific instance of the concept, produces the value of the attribute for the given instance. Figure 3.6 depicts the schema for specifying a domain concept, and the particular schema specifying ROUTER's concept of intersection.

3.3.2.5 The Domain Relation

A domain relation is specified as a tuple $DR := (i, o, tt, p, ip, indx)$. The elements i and o of the *dr-tuple* are the types of the arguments that the relation takes as input and produces as output correspondingly. The elements tt, p, and ip of the *dr-tuple* refer to the actual knowledge of the problem solver regarding this domain relation. This knowledge may have the form of a truth table, $tt(dr)$, which includes all the pairs $(i_k, o_k) \in i(dr)\ Xo(dr)$ for which the relation is true. Alternatively, there may be a predicate, $p(dr)$, which, when applied to any instance $i_k \in i(dr)$,

[2]The terms domain concept and domain object are used interchangeably throughout this thesis.

concept:	the name of the concept, c
domain:	the domain in which its instances take values, dË
attrs:	the set of attributes applicable to this type of concept, $attrs$Ë
identity test:	the predicate evaluating identity between two instances of this type, idË
concept:	intersection
domain:	intersection-domain
attrs:	(name: streets, function: (lambda(x) x), type: (list-of 2 d-street))
identity test:	same-point

Figure 3.6: The domain-concept schema, and the specification of the intersection concept in the SBF language.

returns o_k if $(i_k, o_k) \in dr$ or nil otherwise. In the latter case, the problem solver may also know of the inverse predicate, $ip(dr)$ which when applied to any instance $o_k \in o(dr)$, returns i_k if $(i_k, o_k) \in dr$ or nil otherwise. The difference between these two ways of evaluating dr is that in the first case, the contents of dr are inspectable and easily modifiable. The agent may modify dr by simply adding or deleting tuples in $tt(dr)$. In the latter case, since the predicates $p(dr)$ and $ip(dr)$ are non-inspectable "definitions", the agent cannot modify them.

Finally, the attribute $indx(dr)$ characterizes the type of the domain relation. Domain relations can be of two types: relations whose truth value depends on the state of the world, henceforth called *state-of-the-world* relations, and organizational relations which are "invented" to increase the efficiency of problem solving, henceforth called *convention* relations. This latter type of relations are essentially associations between different types of knowledge, used by the problem solver at the same point in its problem solving, which enable the efficient access of the appropriate pieces of information. An important difference between these two kinds of relations is that the evidence necessary for the agent to modify the organizational relations is less than the evidence necessary to modify the relations whose truth value is derived from the sate of the world. In fact, the problem solver may begin without any specific organization of its knowledge, that is with an undefined domain relation, and it can induce this relation by adding entries into its table on an as-needed basis, for example, every time it fails because of its incomplete organization.

Figure 3.7 depicts the schema for specifying a domain relation, and the particular schema specifying the zone-intersections relation of ROUTER.

3.3.2.6 The Domain Constraint

Constraints are higher-order relations that apply to the objects of a domain. A domain constraint is specified as a tuple $Cnst := (if, then, \{where\})$ where if and $then$ are domain relations. If in some particular domain, there is a constraint $(dr_1(AB)\ dr_2(BA))$, where dr_1 and dr_2 are relations in that domain and A and B are types of objects in that domain, then this constraint means that, if for a particular tuple (ab) the relation dr_1 is true, then the relation dr_2 is true for (ba). If the types of attributes of dr_1 and dr_2 are not the same, it is still possible to specify a constraint between these two relations if there is a predicate $where$ such that it applies to attributes of the relation dr_1 and produces the attributes of dr_2 which do not belong in the set of attributes of dr_1.

Figure 3.8 depicts the schema for specifying a domain constraint, and the particular schema specifying a constraint between the zone-intersections and the zones-of-int relations of ROUTER.

Figure 3.9 depicts the domain of ROUTER that was used in its integration with AUTOGNOSTIC.

relation:	the name of the relation, dr
input arguments:	the types of concepts that it takes as input arguments, $i(dr)$
output arguments:	the types of concepts that it takes as output arguments, $o(dr)$
truth table:	the table enumerating the tuples which belong in the relation, $tt(dr)$
predicate:	the predicate evaluating whether a particular tuple belongs in the relation, $p(dr)$
inverse predicate:	the predicate for deriving the input arguments given the output arguments of a tuple that belongs in the relation, $ip(dr)$
indexing relation:	whether or not the relation is used for indexing purposes only, $indx(dr)$

relation:	zone-intersections
input arguments:	zone
output arguments:	(list-of intersections)
truth table:	zone-intersection-table
predicate:	
inverse predicate:	
indexing relation:	t

Figure 3.7: The domain-relation schema, and the specification of the zone-intersections relation in the SBF language.

constraint:	the name of the constraint, $cnst$
if:	one of the relations this constraint involves, $if(cnst)$
then:	the other relation this constraint involves, $then(cnst)$
where:	a relation specifying how the non-common attributes of relations $if(cnst)$ and $then(cnst)$ relate, $where(cnst)$

constraint:	cnst1
if:	(zone-intersections z? int?)
then:	(zones-of-int int? z?)

Figure 3.8: The domain-constraint schema, and the specification of a constraint between the relations zone-intersections and zones-of-int in the SBF language.

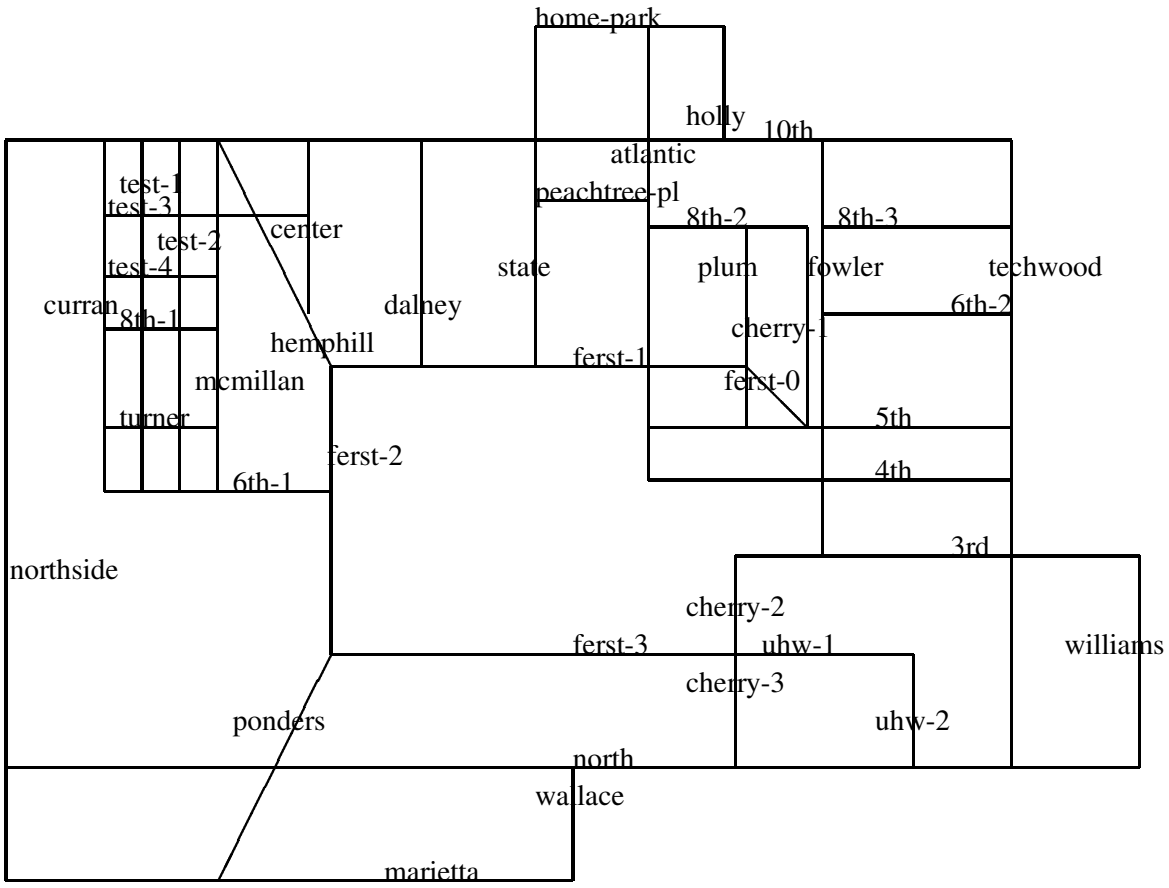


Figure 3.9: ROUTER's navigation space.

CHAPTER IV

INTEGRATING AUTOGNOSTIC WITH A PROBLEM SOLVER

The previous chapter discussed the SBF model of ROUTER, one of the problem solvers with which AUTOGNOSTIC has been integrated. This model was used to provide examples illustrating the use of the SBF language. This chapter presents the SBF models of the other two problem solvers that AUTOGNOSTIC has been integrated with, i.e., KRITIK2 and REFLECS. It also uses the task of modeling REFLECS as an example to illustrate the process of AUTOGNOSTIC's integration with an arbitrary problem solver.

4.1 Kritik2

KRITIK2 [Bhatta and Goel 1992], [Goel 1989], [Goel 1991a], [Goel 1991b], [Stroulia and Goel 1992a], [Stroulia and Goel 1992b] is an autonomous design system that integrates model- and case-based reasoning in the context of adaptive, conceptual design of physical devices.

KRITIK2 has a memory of designs of known devices. Each one of these devices is modeled in terms of KRITIK2's structure-behavior-function (SBF) language (the precursor of AUTOGNOSTIC's SBF language). The SBF model of a device in KRITIK2's memory explicitly represents the functions delivered by the device, its structure, and its internal causal behaviors, that is, the interactions among its components which result in the accomplishment of the overall functions of the device.

Given the specification of the function desired of a new device, KRITIK2 first retrieves from its memory a design that delivers a function similar to the desired function. KRITIK2 organizes the devices in its memory in terms of their functions in a multi-dimensional hierarchy. This organization enables KRITIK2 to efficiently retrieve from its memory a device sufficiently close to the desired one, and therefore relatively easy to adapt towards delivering the new desired function. Next, KRITIK2 determines the differences between the functions delivered by the selected design and the new functions desired of it. It then proceeds to search the SBF model of the retrieved design in order to identify the causes of these functional differences. Once it has identified a set of structural elements in the known design that are potentially responsible for its failure to deliver the desired function, KRITIK2 retrieves one or more repair plans from its repair-plan memory that can eliminate the causes of this failure. The repair plans can be of a wide variety. Some of them, such as the component-replacement plan for example, modify some parameters of the known design. Others, based on case-independent models of generic teleological mechanisms (GTMs) such as cascading and feedback, give rise to more complex, non-local, modifications. Each repair plan is indexed first, by the types of functional differences it can eliminate, and second, by the knowledge conditions under which it is applicable. For any particular problem, several plans may be potentially applicable. The application of a repair plan to the old design results in a candidate design and a revised SBF model for it. KRITIK2 verifies the modifications by qualitatively simulating the new SBF model. If the evaluation fails, KRITIK2 redesigns the candidate design. Otherwise, it proceeds to store the new design and the corresponding SBF model in its memory for further reuse in the future. To identify the appropriate indices under which to store the new design, KRITIK2 uses the SBF model of the new design as a causal explanation of how its structure accomplishes its function. This explanation then identifies the features of the design which are important for its functioning, and these features are used as indices in addition to the design function.

Figure 4.1 diagrammatically depicts the task structure of KRITIK2's reasoning[1]. For a detailed description of AUTOGNOSTIC's SBF model of KRITIK2's design process see Appendix 2. KRITIK2's adaptive method for design decomposes the overall task into two subtasks: retrieval of an existing design, and adaptation of this design to meet the new requirements.

Given a design problem, KRITIK2 probes its memory with the specification of the functions desired of the new device and selects a set of past cases that can be used as the basis for designing the new device. KRITIK2's design memory is hierarchically organized around the different properties of the substances that it knows about, and the ranges of the values that these properties have in each particular device. Thus, to retrieve the set of cases relevant to its current problem, KRITIK2 first identifies the properties mentioned in the new functional specification, new-function's properties identification, and the properties that are used as dimensions in the design-memory hierarchy, known-properties identification. If the intersection of these two sets is empty, i.e., if the properties mentioned in the new functional specification are not used as indexing properties in the memory hierarchy, then KRITIK2 proceeds to elaborate the problem. In the elaboration phase, KRITIK2 attempts to infer other properties that may be relevant to the substances mentioned in the new function, although they are not explicitly mentioned in it. To do that, it uses its conceptual memory of substances. Based on the properties relevant to the new function and the properties used as indices in its memory, KRITIK2 localizes its search for an appropriate past design case to a set of few memory nodes at the second level (i.e., the property level) of the memory hierarchy. This is the refinement along properties task. At this point, based on the values of the properties in the desired function, it further refines its search to a set of nodes at the third level (i.e., the value level) of the memory hierarchy, refinement along values. Finally, KRITIK2 collects the design cases associated with this restricted set of memory nodes, get-cases-of-node, and returns them as the set of cases that can be potentially used to solve the problem at hand. Next, it orders these selected cases in terms of their relative "ease-of-adaptation" and proceeds to adapt the one which is evaluated as the easiest to adapt.

KRITIK2's model-based adaptation method decomposes the adaptation task into four different subtasks: identification of functional differences, blame-assignment, repair, and model assembly. The goal of the first subtask is to compile a list with the differences between the output behaviors of the existing design and the output behaviors desired of the new design. Next, based on the SBF model of the retrieved design, the blame-assignment subtask identifies a list of structural elements of the old design which are responsible for the "failure" of this old design to accomplish the new desired function. The goal of the repair subtask is, given the old design, the desired function, and the set of possible faults, to actually modify the structure of the old design so that it can deliver the required function. KRITIK2 knows several repair plans which it can potentially use to accomplish the repair subtask. The applicability of these plans depends on the type of the difference between the function of the old design and the new desired one. The component-replacement and the structure-replication plans are both applicable when the difference between the two functions lies in the value of some substance property.

The component-replacement plan decomposes the repair task into the following subtasks. First, KRITIK2 identifies one component in the old design which belongs in the list of possible-faults and for which there exists a similar component in KRITIK2's memory with a different parameter appropriate for the new design. The fact that this component (henceforth, the faulty component) belongs in the set of possible faults means that the value of a substance property which differs in the two functions depends (i.e., is either directly or inversely proportional) on this component's parameter. If the parameter of the faulty component affects the value of the property to be modified in a directly proportional manner, and the value of this property in the desired function is greater than its value in the function delivered by the retrieved design (or, if the faulty component affects the value of that property in a inversely proportional manner, and the value of this property in the desired function is less than its desired value) then KRITIK2 probes its component memory for a component similar to the faulty one, but with a higher parameter, component of higher-param retrieval. Otherwise, it searches its component memory for a similar

[1]To date, only the retrieval and adaptation subtasks of KRITIK2 have been modeled. That is, the storage task has not been modeled.

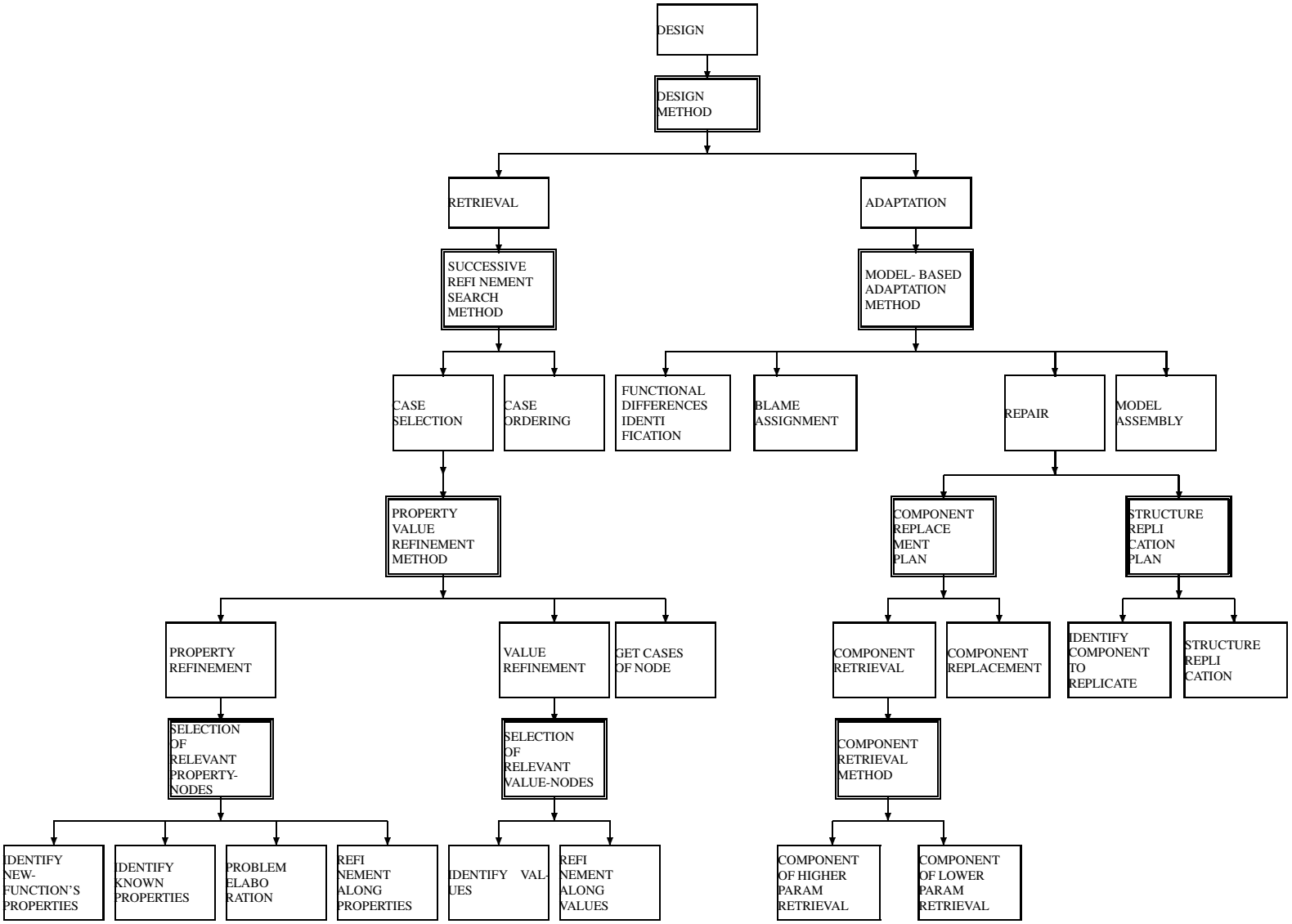


Figure 4.1: KRITIK2's task structure.

component but with a lower parameter, component of lower-param retrieval . Notice that these two tasks are simply two different functional abstractions of the same structural element. That is, there is a single procedure that carries out both of these tasks. If there exists a component with an appropriate parameter then KRITIK2 can replace the component in the old design with the new-component and the resulting device should accomplish the desired function. It is important to notice here that KRITIK2 does conceptual design where the values of the design parameters are only qualitative; therefore, a modification of the "problematic parameter values" in the right direction is evaluated to meet the requirements.

The structure-replication plan sets up two subtasks for the repair task: the identification-of-a-component-to-replicate and the actual replication of this component. The first subtask identifies one component in the old design that belongs in the list of possible faults. If the value of the problem property is directly proportional to this component's parameter and its value in the desired function is higher than its value in the old design (or, alternatively, if the value of the problem property is inversely proportional to this component's parameter and its value in the desired function is lower than its value in the old design) then KRITIK2 can replicate this component to achieve the desired modification.

4.2 Reflecs

The third problem solver with which AUTOGNOSTIC was integrated was a reactive planner. This reactive planner was implemented in the AuRA architecture for robot planning [Arkin 1986]. The result of this integration is a system called REFLECS. The rationale behind the integration of AUTOGNOSTIC with a reactive planner was twofold:

- The reactive planner is a problem solver of a kind very different from both ROUTER and KRITIK2. Where these two problem solvers are deliberative and have a lot of knowledge about their respective domains, the reactive planner is designed in a completely different research paradigm. It does not have any model of its domain. Instead, its planning behavior emerges from the synthesis of a set of primitive reactive behaviors.
- In addition, the redesign problem with which REFLECS was tested was a real problem that actually occurred in the context of the AAAI-93 robotic competition.

The AuRA architecture proposes three levels of planning behavior: *mission*, *navigation* and *pilot*. The *mission* planner takes as input the robot's mission, that is a set of goals such as *copy the paper* and *collect mail* for example. It has the goal of producing as output a plan for achieving these goals, such as *go to the copier room*, *make a copy*, *go to the mail room*, *collect the mail*, and *return to office*. The *navigation* planner is responsible for producing a route for each one of the steps in the mission plan. For example, in order to accomplish the step *go to the copier room* of the above plan, the navigation planner might propose that the robot has to *get out to the corridor*, *turn right*, *traverse the corridor*, *turn left at the end into a second corridor*, and *get into the first door on the right side*. Finally, the *pilot* planner is responsible for producing a sequence of motor actions that can accomplish the routes produced by the navigation planner. At the pilot level, the robot's planning behavior is accomplished reactively. At each point in time, a set of perceptual and a set of motor schemas are active. The perceptual schemas are linked to the robot sensors and thus perceive some aspect of the state of the environment at each point in time. Each motor schema is linked to one or more of these perceptual schemas. Depending on the sensory information these perceptual schemas provide, and on the gain of the motor schema, (i.e., the level of the schema activation), each motor schema produces a vector. The actual motion of the robot at each time depends on the synthesis of all the vectors produced by all the active motor schemas. Figure 4.2 diagrammatically depicts the perception-motion cycle at the reactive level of the AuRA architecture.

It is important to note here that, each time the reactive planner has to perform a new task, a novel configuration of perceptual and motor schemas, which will be sufficient for this new task, may have to be designed. Consider for example the task of "rearranging an office". For this task, the robot is placed in an office with several scattered boxes in it, and it has to move the boxes into a

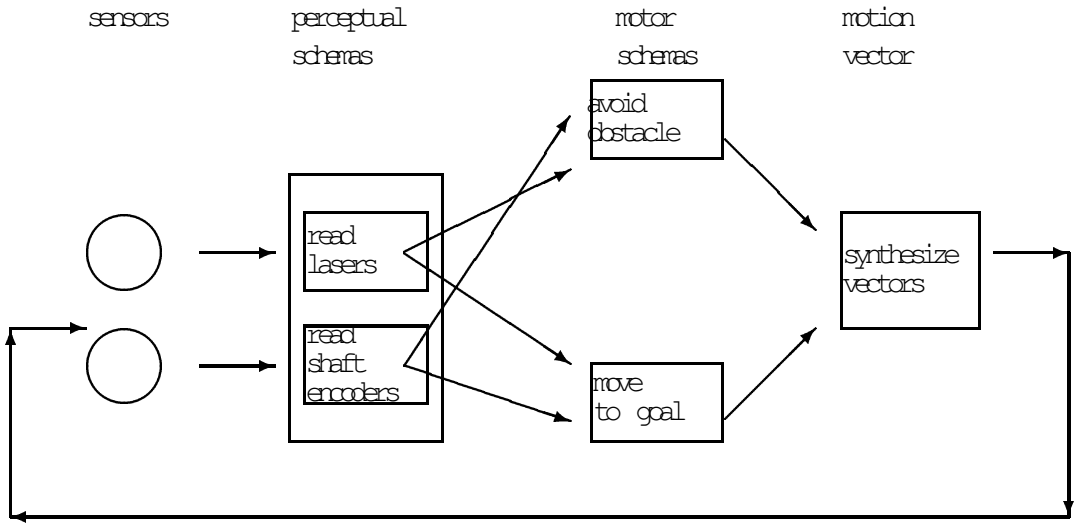


Figure 4.2: The perception-motion cycle at the reactive level of the AuRA architecture.

designated area of the office. To accomplish this task, the planner uses the following process. For each box it has to bring to the designated area, the robot goes to the designated location, then plans its path towards the target box, grabs the box, and brings it back to the designated location.

While the robot is planning its path towards the box, it uses the readings from two different types of sensors, lasers and shaft encoders, which are linked to the read-lasers and read-shaft-encoders perceptual schemas correspondingly. The readings from the shaft encoders indicate the robot's position at each point in time. The lasers' readings indicate the position of the static objects in the immediate environment of the robot. The target box is also included among these static objects, and its position is updated each time a new reading of the lasers is available. There are three motor schemas that are active, i.e., move-to-goal , avoid-obstacle , and synthesize-vectors . The first one uses the readings from the shaft encoders, i.e., the robot's current position, and the position of the target box to produce a vector pushing the robot towards the box. The second one uses the lasers' readings and the robot's current position to produce a vector repulsing the robot from the obstacles around it. The actual movement of the robot is then decided by the synthesis of these two vectors, by the third motor schema, synthesize-vectors . The synthesis is accomplished through vector addition with normalization of the final vector magnitude. The overall task of getting from the designated location to the box is accomplished through a repetition of this perception and motion cycle. The cycle stops when the reading of the shaft encoders shows that the robot has reached its goal, that is, when the robot is within a small radius from the target box.

4.3 Integrating AUTOGNOSTIC with a Problem Solver: The Method

By this point, it should be evident that AUTOGNOSTIC is a shell, rather than a system by itself. In principle, it can be integrated with any problem solver that satisfies some specific requirements (described in section 9.2). When integrated with a particular problem solver, AUTOGNOSTIC can reflect on its problem-solving process in order to adapt it and improve it. In order to integrate AUTOGNOSTIC with some problem solver, AUTOGNOSTIC's user should

1. identify the problem-solver's design elements, and specify their functionality in AUTOGNOSTIC's SBF language,

2. establish the control flow among the primitive design elements, and

3. establish the communication between AUTOGNOSTIC and the problem solver.

Let us now use REFLECS as an example to illustrate the process by which AUTOGNOSTIC is integrated with a particular problem solver.

Identify the problem-solver's design elements and specify their functionality in AUTOGNOSTIC's SBF language

The first step of this process is to identify the design elements comprising the problem solver. In the case of the two deliberative problem solvers, ROUTER and KRITIK2, these design elements are the tasks of the problem solver. In the reactive planning paradigm, the design elements that give rise to behavior are the perceptual and motor schemas of which they consist. Having identified the set of the different design elements, their functionality needs to be specified in terms of their inputs, outputs, and the specification of the transformation they perform between the two. Essentially, for each one of the design elements a task schema, as shown on the top of Figure 3.3, should be instantiated.

The design elements of REFLECS are the perceptual schemas read-lasers and read-shaft-encoders, and the motor schemas move-to-goal, avoid-obstacle, and synthesize-vectors. Notice that the AuRA architecture makes a specific commitment on the "granularity" of the elementary design elements, i.e., the primitive design elements are the schemas. Thus, the process of identifying these elements in a particular system in the AuRA architecture is easier than in the case of an arbitrary system which may have been developed from scratch, and may or may not follow any architectural commitments. In the case of systems such as ROUTER and KRITIK2 the identification of the primitive design elements depends on how the system developer conceptualizes the system. AUTOGNOSTIC's theory at this stage does not provide any guidance on how to go about identifying the right-size design elements of a system. In any case, having identified these primitive elements, their functionality is described in terms of task schemas. In the case of REFLECS the specifications of these schemas in AUTOGNOSTIC's SBF language are shown in Figure 4.4, and are also depicted diagrammatically in Figure 4.3.

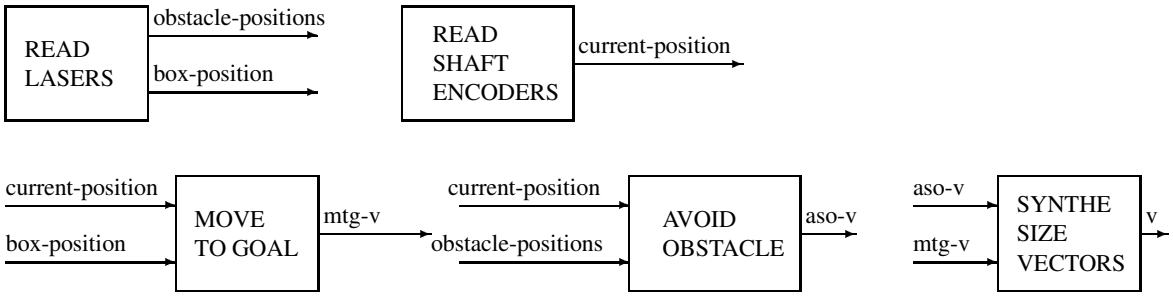


Figure 4.3: The leaf tasks of the reactive-planner's task structure along with their inputs and outputs. These leaf tasks specify the functionality of the primitive design elements of the reactive planner, its perceptual and motor schemas.

AUTOGNOSTIC's SBF model of the problem solver captures the functional and compositional semantics of the problem-solver's task structure. In the case of REFLECS, note that AUTOGNOSTIC's representations of the reactive planner (Figures 4.3, and 4.4) do not completely specify the

task:	read-lasers	task:	read- shaft-encoders
output:	(lasers goal-location)	output:	(actual- location)
procedure:	read-lasers-schema	procedure:	read-shaft- encoders-schema
task:	move-to-goal	task:	avoid-obstacle
input:	(actual-location goal-location)	input:	(actual-location lasers)
output:	(mtg-vector)	output:	(aso-vector)
procedure:	mtg-schema	procedure:	aso-schema
		prototype:	aso-schemas
task:	synthesize		
input:	(mtg-vector aso-vector)		
output:	(synthesis-vector)		
procedure:	synthesis-schema		
under_condition:	(not (diametrically-opposed mtg-vector aso-vector))		

Figure 4.4: The specification of the leaf tasks of REFLECS, in AUTOGNOSTIC's SBF language.

semantics of the perceptual schemas or the semantics of the first two motor schemas. In the case of perceptual schemas, this is because their output depends on the state of the environment alone. Thus there is no input information to which their output can be directly related. In the case of the two motor schemas, this is because their functions are complex mathematical computations. AUTOGNOSTIC's representation of the motor schemas specifies qualitative abstractions of these mathematical computations. For example, for the *synthesize-vectors* schema a condition under which this schema returns a "meaningful" vector is described, and this condition partially captures the functionality of this schema. This condition specifies that this schema produces an output vector only when its input vectors, *mtg-v* and *aso-v*, are not diametrically opposed to each other. The predicate *diametrically-opposed* used to define this condition stands for a mathematical function that computes whether or not the two vectors are diametrically opposed to each other. When AUTOGNOSTIC needs to evaluate the condition, it can apply this predicate to the particular values of the vectors *mtg-v* and *aso-v*. In principle, such a condition could also refer to a domain relation described exhaustively in a truth table. In that case its evaluation would be done through a search in this truth table. This case does not occur in REFLECS because it is a reactive system, and as such it does not have internal knowledge structures. However, it occurs in the integrations of AUTOGNOSTIC with both ROUTER and KRITIK2 and it gives rise to some implementation issues discussed in the next section.

Notice also the slot *prototype* of the *avoid-obstacle* schema: this is a pointer to a family of similar motor schemas, where the functionality of all schemas in this family is to avoid obstacles. Each motor schema in this family is implemented in a different manner, and exhibits a particular variation of the avoid obstacle behavior.

At this stage the information flow among these leaf tasks of the reactive-planner's task structure has also been established through the types of information they consume as input and produce as output. For each type of information mentioned in the Figure 4.4 a corresponding information-type schema has to also be specified, and for each different type of domain object of which each type of information is an instance a corresponding domain-concept schema has to be specified. These specifications are shown in Figures 4.5 and 4.6.

Establish the control flow among the primitive design elements

Having specified the primitive design elements of the problem solver, i.e., the leaf tasks of its task structure, the next step becomes to establish the flow of control among them. This is accomplished by

Types of information:

name: actual-location
type of object: d-location
produced by: (read-shaft-encoders get-status step-in-get-to-box)
input to: (move-to-goal avoid-obstacle)

name: goal-location
type of object: d-location
produced by: (read-lasers get-status)
input to: (move-to-goal)

name: lasers
type of object: d-lasers
produced by: (read-lasers get-status)
input to: (avoid-obstacle)

name: mtg-vector
type of object: d-vector
produced by: (move-to-goal)
input to: (synthesize)

name: aso-vector
type of object: d-vector
produced by: (avoid-obstacle)
input to: (synthesize)

name: synthesis-vector
type of object: d-vector
produced by: (synthesize move)

Figure 4.5: The specification of the types of information in the task structure of REFLECS, specified in AUTOGNOSTIC's SBF language.

specifying the methods that organize these leaf tasks into recursively higher-level tasks, and finally into the overall task of the system. In principle, there are several ways in which these internal problem-solving methods and intermediate subtasks can be conceptualized, and AUTOGNOSTIC at this stage does not provide any guidance on how to explore the alternatives and which among them to select. What is important, however, is that whenever there is a particular ordering among some tasks, this should be specified in the control specification of the method organizing these tasks.

For example, in REFLECS all perceptual schemas have to be invoked before any of the motor schemas that use the sensory input they perceive, and the *synthesize* schema should be invoked after the *move-to-goal* and *avoid-obstacle* schemas have been invoked. The former constraint is captured in the specification of the method *perception-motion method*, which specifies that the *perceive* task should be accomplished before the *move* task. The latter constraint is captured in the method *motor schemas*. The specifications of both these methods in SBF language are shown in the Figure 4.7, and diagrammatically in Figure 4.8. The schemas for the intermediate-level subtasks of REFLECS, *perceive* and *move*, are not shown here. But it is sufficient to mention that their specification includes the methods by which they are accomplished, the input information required by all the subtasks below them, and the output information produced

Types of domain concepts:

```
name: d-location
id test: spatially-close
attribute: ((x-coordinate d-number (lambda(x)(first x)))
            (y-coordinate d-number (lambda(x)(second x)))

name: d-vector
id test: (lambda(v1 v2) (and (same-direction (direction v1)(direction v2))
              (spatially-close (size v1)(size v2))))
attribute: ((direction d-number (lambda(x)(first x)))
            (size d-number (lambda(x)(second x)))

name: d-lasers
id test: equalp
```

Figure 4.6: The specification of the domain concepts of REFLECS, specified in AUTOGNOSTIC's SBF language.

by their subordinate subtasks. The combination of the perceive and move tasks constitute one step in the cycle of steps that accomplishes the task of getting to the box.

```
name: perception-motion-method
applied to: step-in-get-to-box
subtasks: (perceive move)
control: ("serial-op" perceive move)

name: perceptual-schemas
applied to: perceive
subtasks: (read-lasers read-shaft-encoders)
control: ("parallel-op" read-lasers read-shaft-encoders)

name: motor-schemas
applied to: move
subtasks: (move-to-goal avoid-obstacle synthesize)
control: ("serial-op" ("parallel-op" move-to-goal avoid-obstacle) synthesize-vectors)
```

Figure 4.7: The specification of the methods of REFLECS.

Establish the communication between AUTOGNOSTIC and the problem solver
At this point the SBF model of the problem solver with which AUTOGNOSTIC is about to be integrated

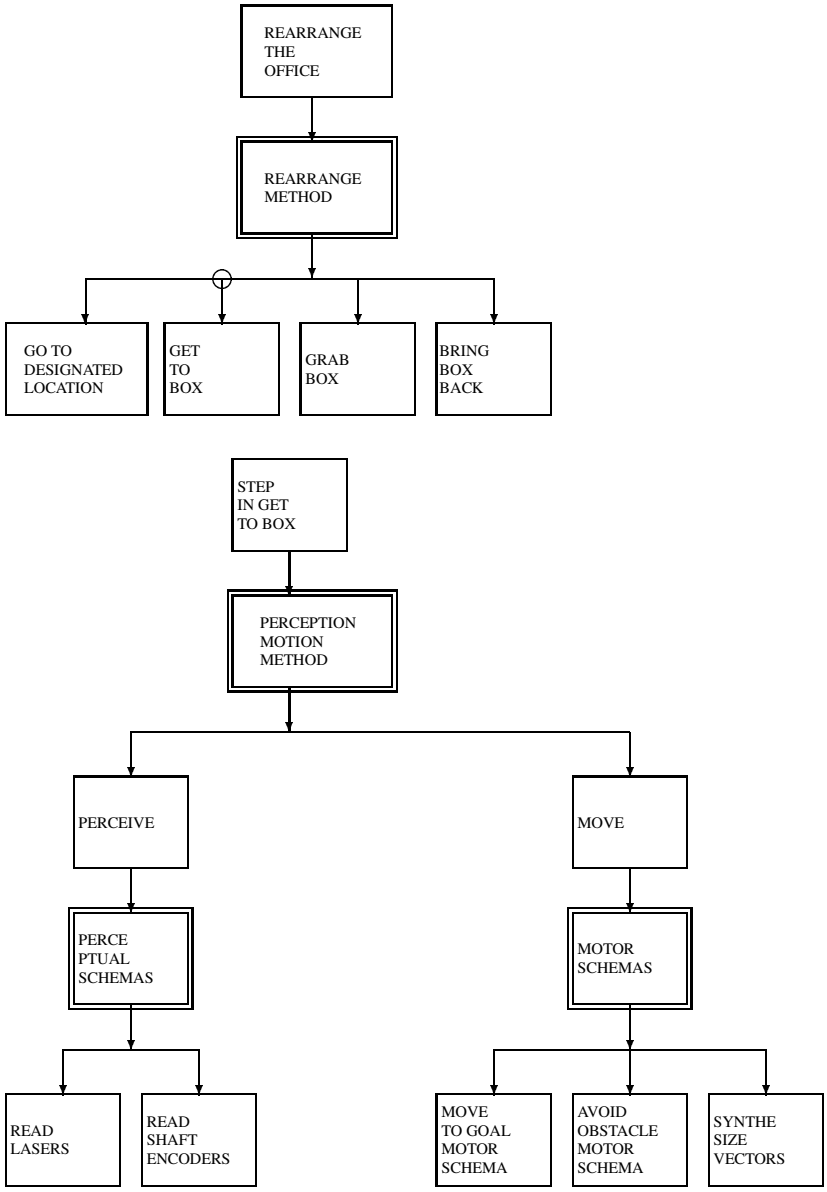


Figure 4.8: The task-structure decomposition of the step-in-get-to-box task.

is complete. The last remaining step is to establish a two-way communication between the problem solver and AUTOGNOSTIC. The communication from the problem solver to AUTOGNOSTIC is required in order to enable AUTOGNOSTIC to access and evaluate the actual behavior of the problem solver as it addresses a particular problem. The communication from AUTOGNOSTIC to the problem solver is required in order to enable AUTOGNOSTIC to modify the design elements of the problem solver.

This communication is implemented in two different ways. In the case of AUTOGNOSTIC's integration with the two deliberative problem solvers, ROUTER and KRITIK2, there was a single source of control for the reflective system as a whole, i.e., both the problem solver and AUTOGNOSTIC. In these integrations, the integrated reflective system exhibits problem-solving behavior

by instantiating the task structure on any given problem. This single source of control invokes the procedures implementing the leaf tasks, receives their data and interprets them according to the semantics of the task structure. This type of communication makes the modification of the problem-solving behavior also straight-forward. Modifications to the model of the task structure directly result in adaptations of the actual problem solving behavior, since this behavior is generated by instantiation of the task structure.

However, this type of communication was not possible in the case of AUTOGNOSTIC's integration with the reactive planner, because it would be completely incongruent with the basic premise of the AuRA architecture. This basic premise is that the actual behavior is not deliberative but emerges from the interaction of a set of active perceptual and motor schemas. Thus, the communication between the two agents, AUTOGNOSTIC and the reactive AuRA planner, was established through an intermediate buffer, implemented as two files. In the first file, the planner wrote the inputs and outputs of each individual active schema for each cycle. This was to establish the communication from the planner to AUTOGNOSTIC. In the second file, AUTOGNOSTIC wrote commands to switch schemas on or off. This was to establish the communications from AUTOGNOSTIC to the planner.

4.4 Some Implementation Issues

The SBF model of a problem solver captures its reasoning process at the knowledge level; it expresses the functional semantics of its tasks, the information and control interactions between them, and their composition into the problem-solver's overall task. That is, the SBF model explicitly represents the functional elements of the problem-solver's design and not how these elements are implemented in the problem solver as a software system. This is because the reflective learning process developed in this thesis, and implemented in AUTOGNOSTIC, is based only on a functional and compositional analysis of the problem-solver's reasoning. However, to be able to effectuate the modifications that it identifies as necessary for the improvement of the problem-solver's performance, AUTOGNOSTIC makes certain assumptions about the actual implementation of the problem solver as a system. These assumptions might necessitate certain modifications to the actual implementation of the problem solver. In particular, two such modifications were necessary, for AUTOGNOSTIC's integration with ROUTER and KRITIK2.

- The elements of ROUTER and KRITIK2 that were responsible for the flow of control among their primitive subtasks were removed.

  When a deliberative problem solver is integrated with AUTOGNOSTIC the two constitute one reflective agent with a virtual problem-solving architecture explicitly represented in the SBF model of its reasoning. For each problem it is presented with, this reflective agent flexibly instantiates the parts of its functional architecture that are pertinent to the problem at hand. The flow of control in this reflective reasoning process is mediated through the SBF model. Thus, there is no need for "hardwiring" a control mechanism in the implementation of the reflective agent.

- Some of the actual data structures in which ROUTER's and KRITIK2's knowledge was represented were modified.

  To be able to modify the problem-solver's actual knowledge (data) when performing domain-knowledge modifications, AUTOGNOSTIC's current implementation makes certain assumptions regarding their implementation in terms of data structures. Thus, a problem-solver's knowledge is represented in two kinds of data structures: lists and associations. For each elementary domain object that the problem knows about, there is a list containing all the instances known to the problem solver; this list is pointed to by the `domain` slot of this object's description in the SBF model. For each domain relation whose truth value depends on the state of the world there is an association table enumerating the tuples of domain objects for which the relation is true: this table is pointed to by the `truth-table` slot of this relation's description in the SBF model.

The original ROUTER system implemented all the different types of knowledge captured by its world model in a single, deeply, nested data structure, and it implemented its path memory as a hash table. In AUTOGNOSTICON-ROUTER the different types of knowledge captured in the original ROUTER's world model are described in terms of the domain lists intersection-domain, zone-domain, direction-domain, and street-domain, and the relation association tables, zone-intersections-table, zones-of-int-table, street-direction-table, children-zones-table. Its path memory, on the other hand, is represented in the paths-of-area association table. Correspondingly, the primitive subtasks that made use of the earlier, more complex data structures were modified to access the ones that replaced them.

Similarly, the data structure for KRITIK2's component and design memory were re-implemented as lists. Furthermore, the hierarchical organization of the memory was explicitly implemented in the following relations: as a relation mapping the memory root to a set of memory nodes, each one associated with a different substance property, root-specialization, as a relation mapping a property memory node to a set of value memory nodes, each one associated with a different value range for the property of its parent node, property-specialization, as a relation mapping a value memory node to a set of value memory nodes, each one associated with a value range subsumed by the value range of its parent node, root-specialization, and as a relation mapping memory-nodes to design cases, indexing-relation.

CHAPTER V

MONITORING

An important issue that intelligent systems face while reasoning about a particular problem is the evaluation of the progress of their reasoning. Intelligent systems have to monitor their reasoning in order to recognize when they have reached their goal, or whether they have failed. The latter role of the monitoring task is especially important since it triggers and sets up the information context for the subsequent learning process.

Most failure-driven learning methods keep a trace of the problem-solver's reasoning [Carbonell *et al.* 1989, Mitchell *et al.* 1981]. This trace alone does not provide information sufficient for the accomplishment of the monitoring task. The trace simply reflects the reasoning steps that the problem solver actually takes, but does not provide any information on the basis of which to evaluate whether these steps are or not appropriate with respect to the problem it is attempting to solve.

> To be able to evaluate its own reasoning, the system needs to have knowledge above and beyond the trace of its actual reasoning.

To recognize the successful completion of its problem-solving process, a system needs a specification of what constitutes a valid, satisfying solution to a given problem. In the problem-solving-as-search view [Carbonell *et al.* 1989, Laird *et al.* 1986, Laird *et al.* 1987], for example, the problem specification consists of an initial state and a set of desired final states. The goal of the problem solver, then, is to reach one of the desired final states, beginning from the initial one. Thus, the problem solver is able to recognize whether it has accomplished its goal or not, by comparing its current state with the desired ones. Alternatively, in the problem-solving-as-recognition-and-adaptation view, [Hammond 1989, Hinrichs 1989, Kolodner 1993], the problem solver has in its memory a set of previously solved problems and their corresponding solutions. Given a new problem, the problem-solver's goal becomes to recognize which of the previous problems is closest to the current one and to adapt its solution to meet the new problem specification. Since the original solutions in the problem-solver's memory are assumed to be correct[1], and since its adaptation strategies are assumed to preserve their internal coherence and correctness[2], the problem solver assumes that it has accomplished its goal when it has completed the adaptation of the retrieved solution. This thesis investigates yet a third approach: The functional semantics of the problem-solver's tasks, as captured in the SBF model of its reasoning, constitute an abstract, problem-independent description of the expected correct behavior of the problem solver. Thus the system can evaluate the success of its reasoning on each particular problem based on whether it conforms with the behavior expected of it.

Failure recognition, however, is a more complicated issue. In the problem-solving-as-search view, the problem solver recognizes that it has failed to accomplish its goal when it reaches an impasse, i.e., a state which does not belong in the set of the final states, from which it cannot proceed any further. Alternatively, in the problem-solving-as-recognition-and-adaptation view, the system may recognize that it has failed if it has never solved a similar problem before, i.e., it does

[1] There may be unsuccessful cases in the problem-solver's memory but they are annotated as such, and they are not used as the basis on which to propose a new solution.

[2] Adaptation in traditional case-based systems is assumed to be local.

not have in its memory a solution that it can reuse, or if it has solved a similar problem previously, but it does not know how to adapt the previous solution to the current situation.

Both these views of problem solving suffer from an important disadvantage: as long as the problem solver has not exhausted the reasoning steps available to it, it is not able to evaluate whether or not it is progressing towards the production of a solution, unless it either reaches the solution or it fails. Sometimes, however, the problem-solving process may flounder and not make any progress towards the solution, or it may seem that it is progressing, although the system has already committed an error which will unavoidably lead to failure. For example, the system may be searching in areas of the problem space far from the locality of the solution, and it may, in fact, be increasing its distance from the desired state. Thus, although it has not reached an impasse, it is not really making progress towards the solution. Alternatively, the system may perform several different adaptations to the solution it retrieved from its memory, but after each adaptation, the retrieved solution may not be any closer to the desired one than before.

To date, there have been several approaches aimed to address these problems. One approach, within the problem-solving-as-search view, adopted by Abstrips [Sacerdoti 1974], is to endow the system with a ranking of the "criticality" of its operators. In this approach, the problem solver can recognize that it is progressing towards the solution, when the operators applicable in its current state are less critical than the operators applicable in the previous one. Another approach, within the problem-solving-as-recognition-and-adaptation view, adopted by Casey [Koton 1988] and by Kritik [Goel 1989], is to endow the system with a model of the domain in which it solves problems. The problem solver can use the model as the basis for evaluating the differences between the current problem and the previously solved one that it is using as the ball-park solution. Thus, the problem solver knows whether it is making progress towards the solution by evaluating the difference before and after each adaptation step. These approaches are domain specific, and although they enable the problem solver to evaluate whether its reasoning process is progressing towards the solution, or is moving far from it, they do not help it to recognize reasoning errors before actually reaching a failed state. The system's comprehension of its own problem solving in terms of the SBF model gives rise to an alternative, domain-independent approach to evaluating its progress and recognizing its errors.

There are four types of knowledge captured in the SBF model that support the monitoring process in a reflective system:

1. For each high-level task that the problem solver can accomplish, the SBF model specifies all the alternative methods that it can use, and for each one of these methods it specifies the subtasks it sets up. Thus, the SBF model outlines all the possible reasoning paths that the problem solver can take to produce a solution for any given problem which is an instance of this task.

2. Given a particular method that the problem solver has selected for a given problem, it knows which tasks are involved in the execution of this method, and to some extent, in what order. Thus, at any point of its reasoning, the problem solver knows what subtasks it has already completed and which ones lie still ahead.

3. For each task that the problem solver can accomplish, the SBF model partially specifies its semantics, i.e., what constitutes a "correct" information transformation for this particular task. Thus, it can evaluate the intermediate solutions it produces for its subproblems against the specifications of the subtasks of which these subproblems are instances. The capability of evaluating intermediate results enables the system to notice errors before its problem-solving process has actually reached a failed state. If it does not have such "standards" for itself, as is the case with non-reflective systems, then it can only assume that its intermediate solutions are correct, until it reaches an overt failure state.

4. Finally, the SBF model specifies the information and control interactions among the subtasks of a method, and the role that each subtask plays (i.e., the information it contributes) in the accomplishment of the overall task. Thus, the system can notice potential discrepancies between the information that a subtask actually produces and the information a later subtask

needs from it, or the information that the subtask in question was expected to contribute to its super-ordinate task, in the service of which it was invoked.

These attributes allow the SBF model to play three roles in monitoring:

1. the functional semantics of the tasks provide a basis for evaluating the correctness of the overall problem-solving process, as well as that of the intermediate results;
2. the rules of task composition into higher-level tasks by methods enable the system to estimate how much of its reasoning lies ahead of it, at any time during problem solving;
3. the expected causal interactions of information passing and control flow among tasks enable the system to recognize instances of pathological task interactions which arise in the context of particular problem-solving episodes.

5.1 The Monitoring Process

The monitoring subtask of AUTOGNOSTIC's reflection process takes as input the *task* that the problem solver integrated with AUTOGNOSTICis asked to accomplish, and the *information context* in which this task is to be accomplished. The information context of the problem task is essentially the initial information presented to the problem solver, i.e., the input information of the problem at hand.

When a new problem (i.e., task + information context) is presented to AUTOGNOSTIC's problem solver, AUTOGNOSTIC first assimilates[3] the given information. The goal of the assimilation process is to evaluate whether the problem input refers to domain objects that AUTOGNOSTIC's problem solver already knows about. If the problem specification is not "understandable", i.e., if there is a reference to some unknown domain object AUTOGNOSTIC halts its reasoning and proceeds to integrate the new information in its domain knowledge before actually proceeding to solve the problem.

The algorithm for monitoring is shown in Figure 5.1. In order to accomplish a given task, t_i, AUTOGNOSTIC first evaluates whether the types of information which this task consumes as input, $i(t_i)$, are available in the current information context (line 4). If all the necessary input information is available, then this task can indeed be accomplished.

However, before actually starting to reason about the task at hand, AUTOGNOSTIC evaluates whether invoking the task at hand will really contribute to the progress of its reasoning (line 5). Quite often the nature of the task structure is recursive, that is, instances of the overall task arise also as its subtasks. Other times, the task structure may be repetitive, that is, a task may be accomplished by the repetition of the same sequence of subtasks. The SBF model of the problem solver makes explicit both these types of control interactions among tasks. Both these types of task structures may give rise to pathological interactions in the context of particular problems. For example, in the former kind of task structures, a particular problem may be invoked in service of itself. In the latter kind, the exact same problem may arise twice in the same repetition sequence. Both these symptoms are indicative of an infinite loop in the process and in such cases AUTOGNOSTIC halts its reasoning.

At this point, AUTOGNOSTIC has established that the input of the task is available and that it is not a pathological repetition of an already accomplished task. Next, AUTOGNOSTIC proceeds to evaluate the conditions of the task, if there are any, and to investigate whether all the types of information that the task will produce as output, $o(t_i)$, are already available in the current information context, and their specific values already meet the qualifications imposed by semantics of the task $s(t_i)$ (line 6). If a task condition is not met or if the expected output is already available in the current information context, then AUTOGNOSTIC determines that it does not need to perform this task. This task is redundant in the current information context, since whatever information this task was meant to contribute to the process of accomplishing the overall problem task, is already known.

[3]The assimilation process is described in detail in Chapter 6

MONITOR($task_k$, $\{(i, v(i))\}*$)

Input:
the task to be accomplished, $task_k$, and
the information context, i.e., a set of pairs, $(i, v(i))$,
where i is an information type, and $v(i)$ is its value in the current context

Output:
a trace of the tasks performed, the methods applicable, and the methods chosen,
the updated information context, with the information produced while problem solving, and
potentially, a failure description.

---

(1) $info\_context$=$\{(i, v(i))\}*$

(2) $trace$=$\emptyset$

(3) $\forall i : (i, v(i)) \in info\_context$ assimilate-value($v(i)$ i)

(4) If $\exists i \in i(task_k) : \nexists (i, v(i)) \in info\_context$

then EXIT(failure-symptom := missing-info($task_k$ i), $trace$, $info\_context$)

(5) If $task_k$ is a subtask of itself in $trace$

$\vee$ there is an exact same sibling subtask in a loop

then EXIT(failure-symptom := recurrent-task($task_k$), $trace$, $info\_context$)

(6) If

$\wedge \forall c \in c(task_k), verified(c, info\_context)$

$\wedge \quad \forall j \in o(task_k), \nexists (j, v(j)) \in info\_context$

$\vee (j, v(j)) \in info\_context \wedge v(j)$ violates $rel \in s(task_k)$

then $trace$=$trace \cup \{task_k\}$

perform($task_k, info\_context, trace$)

(7) **If** $\exists rel \in s(task_k) : false(rel)_{info\_context}$

then EXIT(failure-symptom := failed-semantics($task_k$ rel), $trace$, $info\_context$)

else EXIT(success, $trace$, $info\_context$)

Figure 5.1: The algorithm for Monitoring the problem-solving process.

If the task at hand should indeed be accomplished, i.e., its input is available and its conditions are met, and if it is meaningful to be accomplished, i.e., its output information is not already available, then AUTOGNOSTIC proceeds to perform it. The algorithm for performing a task is shown in Figure 5.2. While performing a task, there are three possible ways in which AUTOGNOSTIC can proceed:

- If the task has associated with it a procedure, $op(t_i)$, AUTOGNOSTIC can invoke the procedure (program code) that accomplishes it (line 1).

  A procedure is a non-inspectable mental action. It returns a set of data which is interpreted as the output information of the task at hand (line 1b) and which is added to the current information context (line 2b).

- If the task at hand is an instance of another known task, AUTOGNOSTIC sets up as its subgoal to accomplish the *prototype* of the task at hand (line 2).

To accomplish a prototype task, AUTOGNOSTIC sets up a new information context which includes the input information of the prototype task which corresponds one to one with the input of the task at hand. AUTOGNOSTIC, at this point, makes the assumption that a prototype task and its instances have corresponding sets of input and output types of information (same number and basically same types). Thus it transfers the values of the input of the instance task as input of its prototype in the new information context. This assumption also enables AUTOGNOSTIC to interpret the output of the prototype task, when it is accomplished, as the output of the task at hand (line 2a), and to extend the information context of the problem-solving process to include this newly produced information (line 2b).

Notice that the information produced in the process of accomplishing the prototype task is not part of the information context of the original problem-solving process. However, in the record produced by AUTOGNOSTIC's monitoring task there is a pointer relating the context of the original process with the context of the process of accomplishing the prototype task (line 2c). The separation of these two records has two advantages. First, it limits the information immediately associated with the record of each problem-solving episode, and consequently it limits the complexity of the blame-assignment task in the case of failure. Second, since the internals of the process accomplishing the prototype task are dissociated from the context in which this task was invoked, then everything that AUTOGNOSTIC may learn to improve this process can be transferred across the several potential uses of the prototype task.

- Finally, if the task has associated with it a set of methods (line 3), $by(t_i)$, which can be applied to it, AUTOGNOSTIC selects one of them, and sets up as its subgoals the accomplishment of the subtasks of the selected method.

  In order to select a method to apply to the task at hand, AUTOGNOSTIC first evaluates their applicability criteria (line 3a) and records which ones of them are applicable in the current context (line 3b). If there are more than one methods whose criteria are met, AUTOGNOSTIC selects one among them arbitrarily (line 3c) and records its choice (line 3d).

  Once a method has been selected, AUTOGNOSTIC refines its current goal, which at this point is to accomplish the task at hand, to be the accomplishment of the selected-method's subtasks. The ordering of the new goals of AUTOGNOSTIC is determined by the control that the method sets up for its subtasks (line 3e). The output information produced by these subtasks further extends the information context of the problem-solving process.

As AUTOGNOSTIC accomplishes a task, in addition to informing the information context of its problem solving, it also records how it accomplishes this task, i.e., by applying a procedure, or by invoking a method to it, or by accomplishing its prototype. From this discussion, it becomes evident that there are two types of failures that AUTOGNOSTIC is able to recognize by itself while monitoring its problem-solving process:

1. AUTOGNOSTIC may find itself unable to complete its problem solving because some of its subtasks requires some type of information that is not available in its current information context.

2. Once, a task is accomplished, AUTOGNOSTIC evaluates whether the specific information it produced as output meets the semantic relations characterizing the information transformation expected by this task. Thus, AUTOGNOSTIC may find that some information does not conform with the semantics of its producing task.

5.2 Examples

This section illustrates the monitoring process of AUTOGNOSTIC with several examples from its integration with ROUTER, and KRITIK2, AUTOGNOSTICONROUTER and AUTOGNOSTICONKRITIK2 correspondingly.

PERFORM($task_k, info\_context, trace$)

Input:
the task to be accomplished, $task_k$,
the information context until now, and
the trace until now.
Output:
the updated trace, and
the updated information context.

---

(1) If $\exists op(task_k)$ (i.e., the task is accomplished by a procedure)

(1a) then $output$ = (apply $op(task_k)$ $\{v(i)\}$)
where $i \in i(task_k) \wedge (i, v(i)) \in info\_context$

(1b) $info\_context = info\_context \cup \{(i, v(i))\}$
where $i \in o(task_k) \wedge corresponding((ioutput(task_k))\ (v(i)\ output))$

(2) If $\exists p(task_k)$ (i.e., the task is an instantiation of a prototype)

(2a) then $output$ = info_context(monitor($p(task_k), \{(i, v(j))\}$))
where $i \in i(p(task_k)) \wedge (j(v(j)) \in info\_context$
$\wedge\ corresponding((ji(task_k))\ (ii(p(task_k))))$

(2b) $info\_context = info\_context \cup \{(i, v(j))\}$
where $(j, v(j)) \in output$
$\wedge\ corresponding((i\ o(task_k))\ (j\ o(p(task_k))))$

(2c) $trace$=$trace \cup \{\rightarrow trace(monitor(p(task_k), \{(i, v(j))\}))\}$

(3) If $\exists by(task_k)$ (i.e., the task is accomplished by methods)

(3a) then $methods_{applicable}$=$\{m\}$
where $m \in by(task_k) \wedge \forall c \in c(m), verified(c, info\_context)$

(3b) $trace$=$trace \cup \{methods_{applicable}\}$

(3c) $method_{chosen}$=$random(methods_{applicable})$

(3d) $trace$=$trace \cup \{method_{chosen}\}$

(3e) $tasks_{to\_do}$=$subtasks(method_{chosen})$
loop **If** $tasks_{to\_do}$=$\emptyset$
then return
else $task_{next}$ = apply($control(method_{chosen}), tasks_{to\_do}$)
perform($task_{next}, info\_context, trace$)
$tasks_{to\_do}$=$tasks_{to\_do} - task_{next}$

Figure 5.2: The algorithm for Performing a task.

5.2.1 Successfully Completed Problem Solving

Example Problem 1: AUTOGNOSTICONROUTER is presented with the problem of route-planning in the following information context: { (initial-point *(10th & center)*) (final-point *(ferst-1 & dalney)*) }. For this problem, AUTOGNOSTICONROUTER successfully completes its reasoning, and produces the path *((center 10th) (10th atlantic) (atlantic ferst-1) (ferst-1 dalney))*.

As shown in Figure 3.1, at the first level of decomposition of the route-planning task, AUTOGNOSTICONROUTER has only one method, route-planning-method. This method sets up four subtasks, classification, retrieval, search and storage. The classification subtask produces *z1* as both initial-zone and final-zone. The retrieval subtask does not return any previously produced path as similar to the current problem. Thus, when AUTOGNOSTICONROUTER gets to the search task, and evaluates the applicability conditions of the case-based and model-based methods only the second one is applicable.[4]

Because both intersections belong in the same neighborhood, AUTOGNOSTICONROUTER invokes the intrazonal-search method next and its subtask search-initialization sets up as its current-path the path *((10th center))*. The next subtask is a repetition of the path-increase subtask until there is a value for the information desired-path. The path-increase subtask is an instance of the task step-in-increase-path task, the task structure of which is depicted in Figure 3.2.

In this particular problem-solving episode, AUTOGNOSTICONROUTER repeats the path-increase subtask six times before actually producing a desired-path. Each time AUTOGNOSTICONROUTER sets up a new information context which is separate from the main information context of the overall route-planning task. For each one of the six repetitions of the step-in-increase-path task, AUTOGNOSTICONROUTER initializes the information context to contain the information possible-paths, final-point and initial-zone which correspond to the input information of step-in-increase-path, current-paths, goal-point and a-zone. When this task is completed, AUTOGNOSTICONROUTER copies the values of the output new-current-paths and the-path back to the main information context as the values of the corresponding information of possible-paths and desired-path. The intermediate information produced while accomplishing the step-in-increase-path task (i.e., current-path and expanded-paths) does not have any equivalent in the main information context. However, it remains available to AUTOGNOSTIC since the main information context, includes pointers to each one of these separate contexts.

5.2.2 Violation of Task Semantics

Example Problem 2: In this scenario, AUTOGNOSTICONROUTER is presented with a problem which reveals a misconception of AUTOGNOSTIC's regarding the functional semantics of the path retrieval task. AUTOGNOSTICONROUTER is presented with the problem of going from *(ferst-1 & hemphill)* to *(home-park & atlantic)*. AUTOGNOSTICONROUTER accomplishes the classification subtask and proceeds to the retrieval subtask, which returns as retrieved-path from its memory *((10th center) (10th atlantic) (home-park atlantic))*. This value for the information retrieved-path does not conform with the semantics of the retrieval subtask.

At this point, it is important to discuss in some detail the functioning of the procedure retrieve-case which implements the retrieval task, and the functional semantics of this task. This procedure searches in ROUTER's memory for a path exactly matching the current problem specification, or if such a path is not available, it searches for a path that begins at the current initial-point, or ends at the current final-point, or begins and ends in the current initial-zone and final-zone respectively. Thus, the it can possibly return a path whose initial-node and final-node belong in the current initial-zone and final-zone

[4] Appendix 1 depicts the complete SBF model of ROUTER, as presented to AUTOGNOSTICONROUTER. The reader may refer to it for more details on the description of ROUTER's tasks, methods, and knowledge.

respectively. The procedure that implements this operator is shown in Figure 5.3. However, the semantics of the *retrieval* task specify that the *initial-node* and the *final-node* of the *retrieved-path* produced by *retrieval* should match the *initial-intersection* and the *final-intersection* in the current information context. This is a situation where the implementation of the procedure does not meet its "design specifications", i.e., the actual behavior of the problem solver does not meet the behavior AUTOGNOSTIC expects from it based on its SBF model. In such situations, AUTOGNOSTIC halts its reasoning and proceeds to assign blame for the failure. The process for this type of failures is discussed in detail in Chapter 6.

```
(defun retrieve-case (source-int dest-int source-map dest-map)
(let* ((cases-matching-on-source-zone
            (find-cases-matching-on-source-zone source-map memory))
       (cases-matching-on-dest-zone
            (find-cases-matching-on-dest-zone dest-map memory))
       (cases-matching-on-both-zones
            (intersection cases-matching-on-source-zone cases-matching-on-dest-zone
            :test #'equalp))
       (cases-matching-on-source-int
            (remove nil (mapcar #'(lambda(x)(if (same-point (initial-node x) source-int) x))
                                cases-matching-on-source-zone)))
       (cases-matching-on-dest-int
            (remove nil (mapcar #'(lambda(x)(if (same-point (final-node x) dest-int) x))
                                cases-matching-on-both-zones)))
       (cases-matching-on-both-ints
            (intersection cases-matching-on-source-int cases-matching-on-dest-int
            :test #'equalp))
       (cases-found (append cases-matching-on-both-ints
                            cases-matching-on-source-int cases-matching-on-dest-int
                            cases-matching-on-both-zones
                            cases-matching-on-source-zone cases-matching-on-dest-zone))
       (case-chosen (car cases-found)))
case-chosen))
```

Figure 5.3: The procedure *retrieve-case* that carries out the *retrieval* task.

5.2.3 Missing Information

Example Problem 3: In this scenario, AUTOGNOSTICONKRITIK2 is presented with a problem that reveals a pathological interaction among the subtasks of a particular method with respect to the information flow among these subtasks. AUTOGNOSTICONKRITIK2 is presented with the problem of designing a nitric acid cooler that cools nitric acid from temperature $T1$ to temperature $T2$, where $T2 << T1$.

AUTOGNOSTICONKRITIK2 retrieves from its design memory a nitric acid cooler which performs a variant of the currently desired function, namely it cools nitric acid from temperature $T1$ to temperature $T2_{old}$ where $T2_{old} << T1$ but $T2_{old} > T2$. Thus, AUTOGNOSTICONKRITIK2 proceeds to adapt the old device to accomplish the new function.

As shown diagrammatically in Figure 4.1 (and in complete detail in Appendix 2), AUTOGNOSTICONKRITIK2 has only one method available to it for accomplishing the *adaptation* task,

i.e., the model-based adaptation method. AUTOGNOSTICONKRITIK2 first accomplishes the first two subtasks of this method, i.e., functional-differences-identification and blame-assignment. The latter produces as potential-faults for the failure of the current device to deliver cooler nitric acid as desired, the insufficient capacity of the water pump of the device, i.e., *(water_pump capacity C)*.

At this point, AUTOGNOSTICONKRITIK2 sets up as its goal to accomplish the repair subtask. As shown in Appendix 2, AUTOGNOSTICONKRITIK2 knows two different methods which can potentially accomplish the repair task, the structure-replication method, and the component-replacement method. Both methods are applicable in the current situation, since their condition, i.e., that func-diffs is a substance-range-difference, is met. In such cases, where more than one method are applicable to the task at hand, AUTOGNOSTIC selects one method arbitrarily, which in the particular episode is the component-replacement method.

The first subtask of this method, i.e., component-retrieval, searches in KRITIK2's memory of elementary components to find a component of the same type as the component included in the potential-faults but with a different parameter. At the time when this problem is presented to AUTOGNOSTICONKRITIK2 its knowledge about the domain does not include a *water_pump* of capacity higher than the one used in the old design. Thus AUTOGNOSTICONKRITIK2 is unable to produce a component suitable to replace the one in the old device. The replace-component task, however, requires as part of its input the replacement component. Thus, at this point AUTOGNOSTICONKRITIK2 fails to continue with the replace-component subtask and therefore to complete its problem solving.

5.3 Summary

AUTOGNOSTIC's monitoring of the problem-solving process goes beyond simply keeping a trace of this process [Carbonell *et al.* 1989, Mitchell *et al.* 1981]. Its comprehension of the purpose of each elementary subtask in the context of higher-level tasks in terms of their functional semantics, and of the information and control interactions among subtasks, enables it to evaluate its progress towards the successful completion of each particular problem-solving episode, and to recognize errors in its reasoning before it has reached a failed state.

Traditionally, AI systems evaluate their progress in solving a particular problem only by recognizing their inability to successfully complete their problem solving. For example, a planner searching in a problem space for a desired state cannot, in general, evaluate whether it is getting closer to the desired state and cannot predict what other states it has to reach before reaching the desired state. It can recognize, however, its lack of progress if it reaches a dead-end state, i.e., a state, which is not the desired one, and from where it cannot proceed anymore because it does not know any operators applicable to it.

Comprehension of the problem-solving process in terms of a SBF model, however, provides AUTOGNOSTIC with a specification of its intended behavior, against which it can evaluate its progress on each problem-solving episode. First, each method indexes all the subtasks which are sufficient to accomplish the overall task; thus, at any point, AUTOGNOSTIC knows what other subtasks are involved in accomplishing its higher-level task. Also, because of its understanding of the role of each subtask in servicing higher-level tasks, AUTOGNOSTIC is able to recognize when a task is redundant; for example, it recognizes that a task is not necessary when the information it produces is already available, or when it is invoked twice in a *loop*, or when it is invoked in service of itself (i.e., infinite recursion). Second, each task sets up expectations regarding the information that it produces; thus for each type of information produced while reasoning about a specific problem, AUTOGNOSTIC has a set of standards (each of the subtasks that produce this information sets up different standards) according to which it can evaluate its progress. For example, AUTOGNOSTIC recognizes problems with its progress when its expectations regarding intermediate information fail, that is when the semantics of a subtask are violated.

CHAPTER VI

BLAME ASSIGNMENT

Blame assignment is a classical problem in artificial intelligence [Minsky 1963, Samuel 1959]. Given a failing problem solver, the goal of the blame-assignment task is to identify the individual decisions that led to the problem-solver's failure to accomplish its task [AI Hhanbook(III)]. Once the causes of the failure have been identified, the learning process can proceed to repair them. Thus, blame assignment is a critical subtask of failure-driven learning.

In general, there are two types of errors in the problem-solver's reasoning which may lead to failures. First, the problem-solver's knowledge about the domain may be incorrect or incomplete, or inappropriately organized and therefore unaccessible, and thus the problem solver may be unable to accomplish its task. Alternatively, the problem-solving process may be incomplete or incorrect: i.e., the problem solver may know all the information relevant to solving the problem at hand, but it may not know how to use it effectively, and thus it may fail to accomplish its task.

6.1 Using a Trace of the Problem Solving to Assign Blame

Earlier work on blame assignment has dealt, to some extent, with both these kinds of causes of failures. Teiresias [Davis 1977] for example, is a system, which guides a domain expert to identify incorrect or missing rules from the knowledge base of Mycin, a diagnostic expert system. Cream [Weintraub 1991] is an autonomous system which identifies incorrect or potentially missing rules in the Qwads expert system (Qwads also performs diagnosis) guided by its understanding of the tasks that Qwads performs. These systems assume that the problem-solving process (i.e., the process of Mycin and Qwads respectively) is correct and the only possible source of error can be the problem-solver's domain knowledge. Thus, the goal of the blame-assignment process, in both these systems, becomes to identify erroneous or missing rules in the problem-solver's knowledge. Respectively, the goal of the learning process becomes to update the problem-solver's knowledge with the correct rules. In both systems the learning process is left to the domain expert.

Alternatively, systems such as Lex [Mitchell *et al.* 1981] and Prodigy [Carbonell *et al.* 1989] assume that the problem-solver's domain knowledge is complete and correct, and also that the problem solver knows all the "primitive" reasoning steps that it can take in that domain (i.e., inferences that it can draw)[1] and therefore, the only possible source of error is the inability of the problem solver to take the right reasoning step at the right time. Thus in both these systems, the goal of the blame-assignment task becomes to identify where in its reasoning process, the problem solver made a wrong selection among its possible reasoning steps, and consequently the goal of the learning task becomes to learn selection heuristics. To explain their failure, and localize the blame for it at some particular point of their reasoning, these systems use a typology of causes of failures described as "pathological patterns" in the problem-solver's trace. As the problem solver reasons about the problem at hand, it keeps a record of the operators it applies. Once the problem-solving process is complete, the trace is a "tree" whose branches are sequences of operators, one of which is successful. At this point, the learner inspects the trace, recognizes in it instances of one or more cause-of-failure patterns, and suggests as a learning task the introduction of selection criteria which will guide the problem solver towards the successful reasoning branch.

[1] The terms "primitive reasoning step", and "primitive inference" are used interchangeably.

In is very important to notice here that the assumption that the problem-solver's set of elementary reasoning steps (operators in the cases of the above systems) is complete and correct, is critical in these trace-based systems. Because of this assumption, they do not have to evaluate, and in fact they do not have any basis for evaluating, whether each individual reasoning step in the trace is correct, that is, whether the results of a particular operator are correct, or whether a new kind of operator is needed. This assumption sufficiently limits the problem of blame assignment, and makes trace-based methods possible. In the domain of Lex, i.e., symbolic integration, and in the domain of Prodigy, i.e., machine-shop scheduling, this may be a realistic assumption. The set of symbolic-integration operators is small, and their use is mathematically established. Similarly, in a closed machine shop, the use of the given set of machines is well established since they are technological artifacts with well known behaviors. This assumption, however, becomes unrealistic when the problem solver moves in more complex domains and starts to perform less well structured tasks. Consider for example, the example of the pedestrian path planner in Chapter 1.[2] This problem solver, having learned to plan paths as a non-driver, formulated the elementary inference *move-to-next-possible-location* as follows: this inference takes as input the current location of the planner and returns as output all the locations that lie on the same streets as the current location, and are adjacent to it. Although this may have been a correct and useful inference for as long as the planner was used by pedestrians, it becomes obsolete and even wrong as soon as the planner is used by drivers. In this new variant of path planning, i.e., planning a path suitable for driving, the system must be able to recognize that its current way of *moving-to-next-possible-location* is wrong, and furthermore, it must be able to fix it. Thus, it has to abandon the assumption that its set of primitive inferences, is complete and correct, and thus trace-based learning methods become insufficient.

6.2 Using a Model of the Problem Solving to Assign Blame

The problem of recognizing that some kind of inferences is incorrectly formulated, or that there is a need for a new kind of inferences, is a very hard and important one, and has counterparts both in AI as a cognitive science and in AI as a design science. From a cognitive-science perspective, humans, very often, engage in new tasks. Children, who just start to solve problems, but also, novice adults, who venture in new problem-solving areas, may often formulate incorrect reasoning strategies. As they solve more problems and gain more experience both of success and of failure, they "debug" and improve these originally imperfect reasoning strategies. From an intelligent-system design perspective, intelligent systems are often employed in environments slightly different than the ones for which they were originally designed. Quite often, in these new environments, the requirements on their problem-solving behavior may be slightly different. Even in the same environment for which the system was originally designed, the requirements may change with the passage of time. In such situations, it is desirable to enable these systems with the capability to redesign their problem solving to meet these new requirements.

Abandoning the assumption of a complete and correct set of reasoning elements makes the trace-based methods insufficient for this, more general, blame-assignment task. The system cannot assume any more that its inferences are correct, simply because they were produced by its reasoning elements. It needs another type of knowledge about its inferences, above and beyond the knowledge necessary to draw them, in order to be able to evaluate their correctness. The question than becomes

> What is the knowledge that a system needs to have, in order to be able to recognize the potentially incorrect functioning of its reasoning elements and the need for new ones?

The hypothesis, explored by this model-based blame-assignment method, is that

[2] In the discussion of the example in Chapter 1, the scenario was that the planner, which was originally intended to be used by pedestrians, was required to produce "drivable' paths.

> the functional semantics of the types of inferences that the problem solver is capable of drawing, i.e., the role they play in problem solving, and the compositional semantics of these inferences, i.e., their "rules" of composition, in the context of its overall problem-solving task constitute sufficient knowledge for evaluating the "correctness" of a specific inference in a particular problem-solving trace.

This hypothesis is inspired by a large body of work in model-based reasoning in the context of diagnosis of physical devices [de Kleer and Brown 1982, Davis 1984, Forbus 1984, Kuipers 1985, Davis and Hamscher 1988]. All of this work is based on the basic premise that "to determine why something has stopped working, it's useful to know how it was supposed to work in the first place" [Davis and Hamscher 1988]. This means that it is useful to understand what are the correct behaviors expected by the device while it functions normally. Since, the correct behaviors expected by the device as a whole are defined by the correct behaviors of its design elements and the interactions between them, it follows that the comprehension of the functionalities of the design elements comprising the device and the interactions between them play a critical role in troubleshooting it when it fails. The above hypothesis is the analog of this statement in the domain of abstract devices, that is, problem solvers.

The inferential capabilities of an intelligent problem solver can only be evaluated in the context of the tasks that this problem solver accomplishes. The path-planner example clearly illustrates this point. The inferential capabilities of the planner are satisfactory as long as it is a pedestrian. However, with respect to the new variant task that it needs to perform, they are not. Because the planner needs to drive its paths, its primitive inference *move-to-next-possible-location* is no longer correct. It is because this inference is used in service of the *"drivable"-route-planning* task in this new scenario, that it is not correct any more, and that it will systematically lead to failures. In addition to understanding the high-level tasks in service of which each particular inference is drawn, the system must also understand the assumptions underlying each one of its inferences. Consider for example the particular situation, where the path produced by the planner includes an "illegal" move, from *(cherry myrtle)* to *(myrtle mapple)*. If the planner does not have any knowledge about its operator, *move-to-next-possible-location*, beyond how to use it, from this failure it can only recognize that it should not use this operator under the condition { current-location =*(cherry myrtle)* AND next-location =*(myrtle mapple)* }. If, however, the planner knows "how the operator works", that is, if it knows that the operator produces as next-location the locations that lie on the same streets as the current-location which are adjacent to it, then it can recognize that, in the current context, the operator is wrongly formulated. At this point, the system may even try to reformulate it correctly, so that it takes into account the legal driving directions of the street on which both the current and next locations lie. In the latter case, the system may be able to learn a new "correct" operator, which will draw the correct inferences in the new context. This type of learning is much more effective than the alternative of explicitly learning all the possible conditions in which the old operator might lead to failure.

The SBF model of a problem solver captures exactly this kind of comprehension of the problem-solver's reasoning process. That is, it explains the overall high-level tasks that the problem solver is able to perform, their decomposition into sets of interacting simpler subtasks by the problem-solver's methods, and the types of domain knowledge that the problem solver has available to it.

6.3 AUTOGNOSTIC's Blame Assignment

As described in Chapter 5, while monitoring, AUTOGNOSTIC can recognize two different types of failure, that is, its inability to complete its reasoning on a particular problem because it is missing some type of information and the production of information which violates the semantics of one of its subtasks. In addition to these types of failures, AUTOGNOSTIC may produce a solution and it may receive as feedback from its environment another solution, preferable to the one it actually produced. In each of these three conditions, a particular blame-assignment method is invoked by

Table 6.1: Blame Assignment: From Failures to Causes.

Conditions of Failure	**Blame-Assignment Suggestions**
Missing Information	Incorrect TS Organization Incorrect Domain Relation Incorrect Knowledge Organization Over-Constrained Task Functionality
Violated Semantics	Task Semantics and Process Mismatch Incorrect TS Organization
Feedback after Completion of Problem Solving	Incorrect Object Representation Unknown Object Incorrect Domain Relation Incorrect Knowledge Organization Over-Constrained Task Functionality Under-Constrained Task Functionality Over-Constrained Method-Selection Criteria Under-Constrained Method-Selection Criteria Missing Method

AUTOGNOSTIC. Table 6.1 describes the different types of causes of failure that AUTOGNOSTIC can potentially identify in each of these failure conditions.

6.4 Assigning Blame for Missing Information

Let us consider the situation where AUTOGNOSTIC has reached a state in its problem-solving process from which it cannot proceed because some type of information necessary for its next subtask is not available (i.e., its value is not specified in its current information context). Figure 6.1 shows the algorithm for assigning blame in the case of missing information.

In such situations, AUTOGNOSTIC's blame-assignment process consists of three subtasks:

1. to identify whether an elementary leaf task that would have produced the information in question has actually been accomplished, and if not, to identify why,

2. to infer a potential value (if it is not given one from its environment) for the missing information and to identify why this value was not produced by the responsible leaf task,

3. finally, if AUTOGNOSTIC cannot find a potential value, it attempts to identify potential errors in the problem-solver's task structure organization which could be blamed for its failure.

AUTOGNOSTIC first identifies the subtask which should have produced the missing information, $t_{problem}$, in the current problem-solving episode. To do that, it uses its SBF model of the problem solver which specifies for each one of the problem-solver's types of information the tasks that produce it, and the record of the problem solving, produced by the monitoring task, which specifies which subtasks were actually accomplished in the current episode. Given these two knowledge sources, AUTOGNOSTIC identifies the last task in the problem-solving episode producing the information in question. There can be several different reasons why this task did not produce the information in question. For example, one reason might be that the method accomplishing the task in question was not correctly carried out in the current problem-solving context, and its elementary leaf subtask which would have produced the missing information was not accomplished. Or alternatively, the problem solver may lack or may be unable to access the domain knowledge that would enable this primitive task to produce the information in question.

The first avenue that AUTOGNOSTIC pursues in its effort to identify the cause for the problem-solver's failure to produce the information i, is to establish whether or not the elementary task responsible for producing the missing information was actually performed. If this task has not been performed (AUTOGNOSTIC uses the problem-solving trace to establish whether a leaf task producing the information in question has been performed) and this was because some of its conditions were not met (line 2), AUTOGNOSTIC postulates that the knowledge based on which these conditions were evaluated was wrong or incorrectly organized (line 2b) and this led to the falsification of the conditions, and consequently to the non-accomplishment of the task. An alternative cause for the non-performance of the task in question might also be that the condition itself is overly specific and excludes situations in which this task should be actually accomplished (line 2c). In that case, the desired adaptation would be to extend the functionality of the task by relaxing the conditions under which it is considered to be useful.

Next, it attempts to infer a possible value for this information. In some cases, the external environment may give AUTOGNOSTIC an appropriate value for the missing information; in such cases the blame-assignment process proceeds as described in section 6.6 (line 3a). Even if however it is not directly given such a feedback from its environment, AUTOGNOSTIC's SBF model of its problem solver enables it to independently infer a potentially appropriate value (line 3b). This capability stems from its comprehension of the the types of knowledge it has available in a particular domain. From the specification of the type of the missing output information, AUTOGNOSTIC knows the type of domain concepts that this type of information refers to. From the specification of this type of domain concept, AUTOGNOSTIC can potentially infer the domain of this object instances, if the domain is specified for that type of concept. It then can search this domain, for a particular instance that meets the semantics of the failed task.[3] If there is such a instance, $v(i)$, then AUTOGNOSTIC proceeds to evaluate the appropriateness of the value it inferred by continuing its problem-solving and monitoring process. If it is able to successfully complete its problem solving, it proceeds to assign blame for the non-production of this desired value $v(i)$ for the information i (line 3b1).

If the system does not have any comprehension of the types of knowledge that are available to it in general, independent of the particular pieces of knowledge that are used during problem solving, then it does not have any means for accessing its knowledge other than the set of reasoning elements it is designed with. And if this set is incorrect or incomplete, then it cannot effectively use its knowledge. If, however, the system has an abstract meta-level understanding of its domain knowledge, such as AUTOGNOSTIC's understanding of the domain concepts and the relations and constraints among them, it has an alternative way to access its knowledge, and shortcut its ill-designed reasoning elements.

If AUTOGNOSTIC cannot find an appropriate value for the missing information (line 3b2), it attempts to reorganize its task structure, in such a way that in similar conditions in the future, the problem solver avoids the problematic task t_{failed}. That is, it infers that the cause of its failure might be its incomplete understanding of the applicability criteria of the lowest-level method (i.e.,

[3] At this point AUTOGNOSTIC actually performs an exhaustive search; in principle, however, this may be computationally a very expensive process. For that reason, it would be better to develop a limited-resources search process.

the method lowest in the task structure in whose task tree the failed task belongs) m that sets up the failed subtask, t_{failed}, (i.e., the subtask which could not be accomplished due to the missing information). AUTOGNOSTIC suggests that the applicability conditions of this method should be restricted, so that when the particular type of information is not available, the method should also be considered inapplicable, and alternative methods should be employed. In such cases, AUTOGNOSTIC essentially infers that it has discovered a pathological interaction of two subtasks of a method which makes impossible the successful completion of the method m. If the task which should have produced the missing information $t_{problem}$ also belongs in $subtree(m, trace)$[4] the revision of the method-selection criteria also implies a reorganization of the task structure so that the task $t_{problem}$ is not part of the method tree anymore.

Example Problem 3: In this scenario, AUTOGNOSTICONKRITIK2 was presented with the problem of designing a nitric acid cooler that cools nitric acid from temperature $T1$ to temperature $T2$, where $T2 << T1$. It retrieved from its design memory another nitric acid cooler which cools nitric acid from temperature $T1$ to temperature $T2_{old}$ where $T2_{old} << T1$ but $T2_{old} > T2$. While trying to adapt the old cooler by substituting its water pump with another one of higher capacity, AUTOGNOSTICONKRITIK2 noticed that there was no appropriate water pump in its component memory and halted its reasoning.

In this situation AUTOGNOSTICONKRITIK2 experiences a failure of the missing-information type. More specifically, as it applies the component-replacement plan to adapting the existing nitric acid cooler to one that will cool the acid over a greater range, it cannot find an appropriate *water_pump* in its component memory, and it fails to perform its component-replacement task since this task requires an alternative component as one of its inputs.

AUTOGNOSTICONKRITIK2 is not able to find in its component-memory and is not given from its environment an appropriate component. Thus it infers that the method component-replacement should be conditioned upon the availability of a proper component to use as a replacement. Notice, AUTOGNOSTICONKRITIK2 suggests the modification of the conditions of this method and not the conditions of the model-based adaptation method, because the component-replacement is the lowest-level method that, if avoided, will make the failed subtask unnecessary for the problem completion, since there is an alternative to that method whose completion does not involve the subtask in question.

In order, however, for the condition not(null(replacement)) to be added to the applicability criteria of the method, the value of this information should be available before the point where the method is considered. Thus the subtask responsible for producing this type of information must be brought outside the method. The actual modification of the task structure is discussed in detail in Chapter 7. Another example of failure due to missing information is discussed in the context of the REFLECS experiment in Chapter 9.

6.5 Assigning Blame for Violated Semantics

Often, in the process of reasoning about a particular problem, AUTOGNOSTIC notices that the value, $v(i)_{actual}$, of some information, i, produced by a task, t_{failed}, violates some semantic relation of this subtask, rel. Recall that the semantic relations of a task specify the correct information transformation that this task is meant to perform in the context of the overall task, i.e., its function in that context. Notice that AUTOGNOSTIC is able to recognize these types of failures because the SBF model explicitly specifies the intended functionality of each task of the problem solver independently from its implementation. Therefore, it admits the possibility that the two do not describe exactly the same information transformation.

If, in the context of a particular problem-solving episode, a semantic relation is violated, there are two possibilities:

[4] $subtree(X)$, where X is either a task or a method, denotes the part of the task-structure subtree with root X. $subtree(X, trace)$ denotes the part of the task-structure subtree with root X which was actually performed in problem solving.

ASSIGN-BLAME-MISSING-INFO ($trace$,$info\_context$,t_{failed},i,$v(i)_{preferred}$)

Input:
the trace of the failed problem-solving episode, $trace$,
the information context of the failed problem-solving episode, $info\_context$,
the task which failed due to missing information, t_{failed}
the missing information, i, and
the value $v(i)_{preferred}$ that i should have been assigned.

Output:
a set of possible causes $FailureCauses_{potential}$

---

(1) $t_{problem} := t$ where $i \in o(t) \wedge t \in trace$
(2) If not(leaf-task($t_{problem}$))
then $t_{missing} := t \in subtree(t_{problem}) \wedge i \in o(t) \wedge$ leaf-task(t)
(2a) **If** $\exists rel \in c(t_{missing})$
(2b) **then If** $\exists tt(rel)$
then Include in $FailureCauses_{potential}$
If $indx(rel)$ **then** INCORRECT-KNOWLEDGE-ORGANIZATION
else INCORRECT-DOMAIN-RELATION
include ($v(i)\ actual\ v(j)_{actual}$) in $tt(rel)$
(2c) **else** Include in $FailureCauses_{potential}$
OVER-CONSTRAINED-TASK-FUNCTIONALITY
modify rel to be true in similar situations
or modify the tasks producing the information of rel so that it is true
(3) If $not(null(v(i)_{preferred}))$
(3a) then $FailureCauses_{potential} :=$
assign-blame-wrong-value(i,nil,$v(i)_{preferred}$,overall-task($trace$),$trace$,$info\_context$)
(3b) else $v(i)_{preferred}$:= INFER($t_{problem}$)
(3b1) **If** $not(null(v(i)_{preferred}))$
then Complete Problem Solving and Monitoring
If success
then $FailureCauses_{potential} :=$
assign-blame-wrong-value
(i,nil,$v(i)_{preferred}$,overall-task($trace$),$trace$,$info\_context$)
(3b2) **else** Include in $FailureCauses_{potential}$
INCORRECT-TS-ORGANIZATION(m,i,t_{failed})
Condition m on availability of i to avoid t_{failed} in similar situations
where $m \in trace$
$\wedge\ t_{failed} \in subtree(m, trace)$
$\wedge\ \nexists\ m' : m' \in subtree(m, trace) \wedge t_{failed} \in subtree(m', trace)$

Figure 6.1: The algorithm for Assigning Blame for Missing Information.

1. either the task is correctly formulated (i.e., its semantics are correct) but its implementation is not and therefore the particular values produced will eventually lead to failure, or

2. its semantics are wrong but its implementation is correct, and therefore the particular values produced are also correct.

In the former case, AUTOGNOSTIC attempts to identify what the correct values should have been for the information that violates the semantics of its producing task, and to infer the potential causes that prevented its production. In the latter case, AUTOGNOSTIC infers that its SBF model of the problem solver may be incorrect and attempts to reformulate its conceptualization of the task whose semantics have been violated by "legal" values. The algorithm for assigning blame for the violation of the semantics of some subtask is shown in Figure 6.2, and is analyzed in greater detail below.

To investigate the possibility that it does not sufficiently "understand" the role of this subtask in the context of accomplishing the overall problem task, AUTOGNOSTIC continues its reasoning on the particular problem (line 1). If the output information produced by the overall task is acceptable, i.e., if the overall solution conforms with the semantic relations of the overall task, AUTOGNOSTIC concludes that the failing semantic relation is incorrect, and this is the reason why it is in conflict with the successful behavior of the problem solver (line 2a). In such a case, AUTOGNOSTIC suggests as an appropriate modification the redefinition of the failed semantic relation so that the particular values of the task input and output in this problem do not violate it. Essentially, AUTOGNOSTIC suggests that the failed relation should be substituted by another, for which the relevant values in the current problem-solving episode would be a positive instance.

If, on the other hand, the output information of the overall task is unacceptable, AUTOGNOSTIC infers that the process carrying out the failed task is indeed erroneous. If AUTOGNOSTIC's environment shows an alternative value for the information whose actual value in the problem-solving episode causes the failure of the semantics, then its goal becomes to assign blame for the non-production of this alternative value (line 2b1). Otherwise, it attempts to identify an alternative strategy for solving the problem which would have avoided the offensive task (line2b2).

Finally, if in spite of these efforts, the blame-assignment process has not identified a single potential cause for its failure, then AUTOGNOSTIC infers that the organization of its task structure might be incorrect, and that the applicability conditions of the method m, in whose subtree the offensive task belongs, should be updated to include the conformity of the value $v(i)$ with the semantic relation rel (line 3).

Example Problem 2: In this scenario, AUTOGNOSTICONROUTER was presented with the problem of going from *(ferst-1 & hemphill)* to *(home-park & atlantic)*. When AUTOGNOSTICONROUTER is presented with this problem, its path memory contains the path *((10th center) (10th atlantic) (home-park atlantic))* which begins from the neighborhood *z1* which is the initial-zone in the current information context. Because there is no better path available, the retrieve-case procedure returns this path as middle-path. This path violates one of the semantics of the retrieval subtask, which specify that the middle-path should begin at the current initial-location. Thus, AUTOGNOSTICONROUTER notices a failure of what it expects the correct behavior of the retrieval task to be. This causes AUTOGNOSTICONROUTER to interrupt its reasoning and try to "understand" this mismatch.

While assigning blame for its failure, AUTOGNOSTICONROUTER completes its reasoning and produces as a solution the path, *((ferst-1 hemphill) (hemphill 10th) (10th center) (10th atlantic) (home-park atlantic))* which meets the functional semantics of the overall route-planning task. Since the solution produced to the overall task is acceptable, although some of the intermediately produced types of information was not, AUTOGNOSTIC infers that its current comprehension of the role that the retrieval task plays in the context of the route-planning task is not valid. To improve that comprehension, AUTOGNOSTIC suggests as an appropriate modification, the substitution of the failed semantic relation, same-point(initial-intersection initial-node(middle-path)), with a new one which will unify the seemingly "erroneous" product of retrieval in the current problem-solving episode with its previous experiences by

ASSIGN-BLAME-VIOLATED-SEMANTICS ($trace$,t_{failed},rel,i,$v(i)_{actual}$,$v(i)_{preferred}$)

Input:
the trace of the problem-solving episode
the task whose semantics were violated, t_{failed}
the violated relation, rel,
the offensive information, i,
the value of the information, $v(i)_{actual}$, and
its preferred value $v(i)_{preferred}$.

Output:
a set of possible causes $FailureCauses_{potential}$

---

(1) Complete Problem Solving and Monitoring

(2) If success

(2a) then Include in $FailureCauses_{potential}$
TASK-SEMANTICS-AND-PROCESS-MISMATCH
DISCOVER(rel') where $true(rel'(v(i)_{actual}))$
SUBSTITUTE rel with rel' in $s(t_{failed})$

(2b) else If $not(null(v(i)_{preferred}))$

(2b1) **then** $FailureCauses_{potential}$:=
assign-blame-wrong-value(i,$v(i)_{actual}$,$v(i)$,overall-task($trace$),$trace$,$info\_context$)

(2b2) **else** $FailureCauses_{potential}$:=
assign-blame-in-alt-ts($i, v(i)_{actual}, v(i)_{preferred}, t_{problem}$)

(3) If the set $FailureCauses_{potential}$ is empty
then Include in $FailureCauses_{potential}$
INCORRECT-TS-ORGANIZATION
Condition m on conformity of $v(i)$ with rel to avoid t_{failed} in similar situations
where $m \in trace$
$\wedge\ t_{failed} \in subtree(m, trace)$
$\wedge\ \nexists\ m' : m' \in subtree(m, trace) \wedge t_{failed} \in subtree(m', trace)$

Figure 6.2: The algorithm for Assigning Blame for Violation of task semantics.

identifying a set new semantic relation which can cover all of them. The process of identifying an appropriate relation, and redefining the semantics of a task is discussed in Chapter 7.

6.6 Assigning Blame for Producing an Unacceptable Solution

Even if, however, the problem-solving process has been completed and a solution to the given problem has been produced, this solution may be incorrect or unsatisfactory. To address this possibility, when AUTOGNOSTIC completes its problem solving, it receives feedback from its environment regarding the solution it produced. If the solution is acceptable, AUTOGNOSTIC proceeds to the next problem. If it is not, AUTOGNOSTIC receives as feedback the desired solution, and proceeds to assign blame for the production of the "wrong" solution. It is important to notice here that in such cases of failure, where the problem solving has been completed and has produced a

solution unacceptable to its environment, AUTOGNOSTIC expects as feedback an alternative solution that would have been acceptable. A simple "yes" or "no" feedback cannot be very helpful to AUTOGNOSTIC, except that it can try to pursue alternative possible strategies to solve the problem. The focus of this thesis, is to investigate how a more informative feedback, i.e., the "preferred solution", can help the system improve its performance.

AUTOGNOSTIC's method for assigning blame for the production of a solution not acceptable by its environment, consists of three major subtasks. The first subtask, *assimilation of feedback information*, investigates whether the feedback solution[5] was not produced because it includes references to objects and relations in the domain that are unknown to the problem solver or are in conflict with the problem-solver's current domain knowledge. If this is the case, i.e., if the problem solver ignores elements of the feedback solution, then it can not possibly produce it, unless it first corrects its domain knowledge.

The second subtask, *blame assignment within the problem-solving strategy used*, investigates whether the strategy used in the failed problem-solving episode, could have produced the feedback under different domain-knowledge conditions, or with slight modifications to the way these subtasks were performed. Often, even though the problem solver knows all the elements of the feedback and the strategy it has used can potentially produce it, that is, the feedback is within the class of solutions that the strategy is designed to produce, it may still fail. This can be because some piece of domain knowledge on which some intermediate inference relies is incorrect or missing, or alternatively, some of the tasks involved in the strategy are under-specified and allow several possible inferences, some of which may lead to solutions other than the desired one.

The third subtask, *exploration of alternative strategies*, investigates whether an alternative strategy should have been more appropriate for the problem at hand. Often, the second subtask may be unable to identify an incorrect or missing piece of domain knowledge which can be blamed for the failure, and it may conclude that the only way that the strategy used could have produced the desired feedback would be to "redefine" the role of the tasks involved in this strategy, that is, to extend their functionality to allow a different, wider class of mappings from their input to their output. In these cases, the problem solver may explore the possibility that alternative reasoning strategies may be more appropriate for solving the problem at hand. The high-level algorithm for this blame-assignment task is shown in Figure 6.3.

6.6.1 Assimilation of Feedback Information

AUTOGNOSTIC's feedback-assimilation process uses the model of the problem-solver's domain knowledge to "parse" the desired solution in order to elicit all the information it contains about the domain. If some of this information is not part of, or is in conflict with, the problem-solver's knowledge, then this is evidence that the feedback does not belong in the domain of solutions that the problem solver can produce. The goal of the assimilation process is to identify such errors in the domain knowledge. Figure 6.4 shows in detail the assimilation process algorithm.

The SBF model (see Chapter 3) of the problem solver includes a general description of the "ontology" of the domain of the problem solvers. For each different concept that the problem solver knows about, the SBF model describes its attributes, the relations that are applicable to it, a predicate for testing identity among its instances, and a pointer to the set of instances of this object that are currently known to the problem solver, i.e., its domain. For each attribute of the concept, the description includes a function for inferring its value from each instance, and the type of the resulting value (line 2). The assimilation process examines whether the instance presented as feedback to the problem solver belongs in the corresponding concept domain (line 1). If not (line 1a), it suggests the integration of new knowledge regarding this instance with its existing domain knowledge, and infers its attributes to further examine whether their values are known and whether they agree with their corresponding expected types.

If at any point, the assimilation process faces an inconsistency between the feedback and its expected attributes (line 3), it considers this as evidence that a change in the representation

[5]The term feedback denotes the type of information for which the problem solving produced the unacceptable solution and the desired solution.

ASSIGN-BLAME-WRONG-VALUE (i,$v(i)_{actual}$,$v(i)_{preferred}$,t,$trace$,$info\_context$)

Input:
the type of information for which the wrong value was produced, i,
the value $v(i)_{actual}$ produced for i,
the value $v(i)_{preferred}$ which should have been produced for i,
the task responsible for producing the information i, t,
the trace of the problem solving, $trace$, and
the information context, $info\_context$.

Output:
a set of possible causes $FailureCauses_{potential}$

---

(1) $FailureCauses_{potential}$:=
assimilate-value($v(i)_{preferred}$,i)

(2) $FailureCauses_{potential}$:= $FailureCauses_{potential}$ ∪
assign-blame-in-ts-used-for-ps(i,$v(i)_{actual}$,$v(i)_{preferred}$,overall-task($trace$),$trace$,$info\_context$)

(3) If ∃ mod ∈ $FailureCauses_{potential}$
∧ type(mod) = OVER-CONSTRAINED-TASK-FUNCTIONALITY
then $t_{problem}$:= the task whose functionality has to be extended
$FailureCauses_{potential}$:= $FailureCauses_{potential}$ ∪
assign-blame-in-alt-ts(i,$v(i)_{actual}$,$v(i)_{preferred}$,$t_{problem}$)

Figure 6.3: The algorithm for Assigning Blame for producing an unacceptable solution.

framework of the problem-solver's domain knowledge may be necessary, in order for the new instance to meet the ontological commitments of the system. An inconsistency between an instance and the object's representation may occur if the assimilation process fails to infer the value of some attribute of the corresponding domain object in the new instance, or the incompatibility between value of the attribute and its expected type. Note that it is because the system has an explicit understanding of its ontological commitments that it can recognize when they fail to cover a new instance. When the assimilation process notices the insufficiency of the representation scheme to express a new instance of some domain object, it also suggests that modifications may be necessary to any subtasks of the problem solver whose semantic relations refer to this attribute.

Example Problem 4: AUTOGNOSTICONROUTER is presented with the problem of going from *(8th-1 & mcmillan)* to *(north & cherry-3)*. For this problem it produces the path *((8th-1 mcmillan) (mcmillan test-3) (test-3 hemphill) (hemphill ferst-2) (ferst-2 ferst-3) (ferst-3 cherry-3) (cherry-3 north))*. At the end of its problem solving, AUTOGNOSTICONROUTER is presented with an alternative solution, preferable to the one that it actually produced, i.e., *((8th-1 mcmillan) (mcmillan 6th-1) (6th-1 ferst-2) (ferst-2 ferst-3) (ferst-3 cherry-3) (cherry-3 north))*.

AUTOGNOSTIC's assimilation process uses the specification of the concept path in the SBF model of ROUTER's path planning to "understand" the feedback. This description (see Appendix 1) specifies that a path is a sequence of "nodes" each of which is an intersection. Thus, the assimilation process uses its general description of intersections to assimilate the specific intersections that constitute the feedback path. Intersections are elementary objects in ROUTER's domain knowledge and all the instances of intersections known to ROUTER are enumerated in the *intersection-domain*. By investigating the contents of the intersection domain, the assimilation process finds that the intersections *(mcmillan 6th-1)* and *(6th-1 ferst-2)* do not belong in it. Moreover, in its effort to

ASSIMILATE-VALUE($v(i)_{preferred}$, i)

Input:
the value $v(i)_{preferred}$ should have been produced for i, and
the type of information, i.
Output:
a set of possible causes $FailureCauses_{potential}$

(1) If $\exists\ d(wo(i))$, (i.e., the $wo(i)$ is enumerated)
(1a) then, if $v(i) \notin d(wo(i))$
then Include in $FailureCauses_{potential}$
UNKNOWN-OBJECT
include $v(i)$ in $d(wo(i))$
(2) $\forall a \in attrs(wo(i))$
assimilate-value (f(f(a), v(i)) type(a))
(3) If the value $v(i)_{preferred}$
does not have all the attributes $attrs(wo(i))$
OR their corresponding values have not been assimilated
then Include in $FailureCauses_{potential}$
INCORRECT-OBJECT-REPRESENTATION
to accommodate the absence of some attribute
or the discrepancy between the value and the type of an attribute
Investigate the operation of the subtasks whose semantic relations
refer to the attribute which is to be modified

Figure 6.4: The algorithm for Feedback Assimilation.

assimilate *(mcmillan 6th-1)* and *(6th-1 ferst-2)*, the assimilation process notices that *6th-1* does not belong in the street domain. Therefore, the assimilation process infers that the cause of the failure is the problem-solver's ignorance of street *6th-1* and its intersections. Therefore it suggests as modifications that could potentially enable ROUTER to produce the desired path, the incorporation of *6th-1* in the street domain, and the incorporation of *(mcmillan 6th-1)* and *(6th-1 ferst-2)* in the intersection domain.

Let us now consider AUTOGNOSTIC's feedback-assimilation process for this same problem with a slightly different feedback from the environment, i.e., *((8th-1 mcmillan) (mcmillan 6th-1) (6th-1 ferst-2) (ferst-2 ferst-3 ponders) (ferst-3 cherry-3) (cherry-3 north))*. While assimilating *(ferst-2 ferst-3 ponders)* as an intersection, AUTOGNOSTIC faces an inconsistency between the type of the attribute `streets` of the `intersection` concept and its value in the specific intersection. That is, while the `streets(intX?)` is a list of two elements, each of which is a street, *(ferst-2 ferst-3 ponders)* is a list of three elements. At this point, the assimilation process has noticed a discrepancy between the ontological commitments of the representation scheme and a particular new instance of a domain object, and therefore it suggests a modification to the representation of the attribute `streets` of intersections in ROUTER's domain knowledge.

6.6.2 Searching for Errors in the Strategy used for Problem Solving

If the assimilation process proceeds without identifying any potential causes for the failure of the problem solver, i.e., all the information conveyed by the feedback solution is already known to the problem solver, AUTOGNOSTIC's blame-assignment process starts to investigate the strategy which was used during problem solving, in order to identify modifications which can potentially enable this same strategy to produce the feedback solution. Figure 6.5 shows in detail the algorithm for blame assignment within the strategy used for problem solving.

It is possible that the desired solution is within the class of solutions that this problem-solving strategy can produce but it was not actually produced because

1. some piece of domain knowledge is incorrect or missing;

   The successful assimilation of the feedback only means that the constituent elements of the solution are known to the problem solver. Still, there may be types of knowledge used to draw intermediate inferences, for which the problem-solver's domain knowledge is incomplete, or incorrect. In such cases, the problem solver may draw an incorrect inference which may lead to an unacceptable solution to the overall problem.

2. some piece of domain knowledge is incorrectly organized;

   Problem solvers organize their knowledge so that they can access it efficiently when they need it. This way, they avoid expensive search at the time of use. However, if the organization is not appropriate, they may sometimes miss useful information which they have but cannot access.

3. finally, the problem-solving strategy is under-specified and allows the production of several solutions to the same problem, including the actual one and the feedback.

AUTOGNOSTIC's blame-assignment process begins at the level of the highest-level task in the task structure whose output is the information for which the problem solver produced an undesirable value, i.e., overall-task($trace$). At this point, AUTOGNOSTIC evaluates whether the feedback solution is within the class of solutions that this task produces for the problems it is given as input (line 1). The functional semantics of the task under inspection characterize exactly this class of solutions, thus if the feedback verifies these relations, then AUTOGNOSTIC infers that it could have been produced by the task in question. Therefore, the blame-assignment process infers that the reason why this value was not actually produced must lie within the internal mechanism of the task, that is, it must be due to some of the subtasks which were performed to accomplish the task under inspection. From the trace of the problem solving, AUTOGNOSTIC infers which method was actually used to solve this task in the current problem-solving episode, and, it proceeds to focus the assignment of the blame to the subtask which produced the undesired value (line 7). If the task under inspection is an instance of a prototype task, then the blame-assignment process moves to assign the blame for the production of the undesired value to the performance of the prototype (line 6).

Notice that without an abstract comprehension of the nature of the information transformation accomplished by the task at hand, the only way for the system to evaluate whether a solution is producible by its problem-solving process is to actually try to produce it (as do Lex and Prodigy, for example), and this can be a very expensive process. Furthermore, if the problem solving process cannot produce the feedback, and if the domain is sufficiently complex, the effort may continue for a very long time.

If at some point, the semantic relations of a task are not satisfied by the input of this task and the feedback (line 2), then the blame-assignment process attempts to infer alternative input values[6] which would satisfy the failing semantic relations (line 3). Essentially, if the desired output cannot be produced by the input that the task actually received in the failed problem-solving episode,

[6]When the task input information is not part of the overall problem specification, that is, the types of information for which alternatives are sought are not part of the givens of the problem.

ASSIGN-BLAME-IN-TS-USED-FOR-PS (i,$v(i)_{actual}$,$v(i)_{preferred}$,t,$trace$,$info\_context$)

Input:
the type of information for which the wrong value was produced, i,
the value $v(i)_{actual}$ produced for i,
the value $v(i)_{preferred}$ which should have been produced for i,
the task responsible for producing the information i, t,
the trace of the problem solving, $trace$, and
the information context, $info\_context$.
Output:
a set of possible causes $FailureCauses_{potential}$

---

(1) CR_{false}=$\{rel\} : rel \in s(t) \wedge rel : i \rightarrow i_{in} \in i(t) \wedge false(rel(v(i)_{preferred}, v(i_{in})_{actual}))$
(2) If $CR_{false} \neq \emptyset$
then $\forall rel \in CR_{false}, rel(i, i_{in}) : i_{in} \in i(t)$
(3) **If** $i_{in} \notin i$(overall-task($trace$))
then INFER alternatives $v(i_{in})_{preferred}$ such that $true(rel(v(i)_{preferred}, v(i_{in})_{preferred}))$
$infos_{alt}$=$\{i_{in}\} : v(i_{in})_{preferred} \neq v(i_{in})_{actual}$
If $infos_{alt} \neq \emptyset$
(3a) **then** $\forall i_{in} \in infos_{alt}$
assign-blame-wrong-value
(i_{in},$v(i_{in})_{actual}$,$v(i_{in})_{preferred}$,t',$trace$,$info\_context$)
(3b) **else If** $\exists$ $tt(rel)$
then Include in $FailureCauses_{potential}$
If $indx(rel)$ **then** INCORRECT-KNOWLEDGE-ORGANIZATION
else INCORRECT-DOMAIN-RELATION
include ($v(i)_{preferred}$ $v(i_{in})_{actual}$) in $tt(rel)$
Include in $FailureCauses_{potential}$
OVER-CONSTRAINED-TASK-FUNCTIONALITY
extend task to produce $v(i)_{preferred}$ instead of $v(i)_{actual}$, for $v(i_{in})_{actual}$
(4) **else If** $\exists$ $rel \in s(t) : \exists$ $tt(rel) \wedge t$ is a leaf task
then Include in $FailureCauses_{potential}$
If $indx(rel)$ **then** INCORRECT-KNOWLEDGE-ORGANIZATION
else INCORRECT-DOMAIN-RELATION
exclude ($v(i)$ $actual$ $v(i_{in})_{actual}$) from $tt(rel)$
Include in $FailureCauses_{potential}$
UNDER-CONSTRAINED-TASK-FUNCTIONALITY
refine task to produce $v(i)_{preferred}$ instead $v(i)_{actual}$ for i
(5) If task t is a leaf task, **then** Exit
(6) If $\exists$ $t' : t'$=$p(t)$ **then** assign-blame-wrong-value(i,$v(i)_{actual}$,$v(i)_{preferred}$,t',$trace$,$info\_context$)
(7) If $\exists$ $by(t)$ **then** assign-blame-wrong-value(i,$v(i)_{actual}$,$v(i)_{preferred}$,t',$trace$,$info\_context$)

Figure 6.5: The algorithm for Assigning Blame within the strategy used for problem solving.

then AUTOGNOSTIC attempts to postulate some alternative potential inputs and thus trace the blame backward to earlier tasks which should have produced it. If there is some intermediate type of information, i_{in}, for which an alternative value, $v(i_{in})_{preferred}$, can be inferred different from the value actually used in the problem-solving process, $v(i_{in})_{actual}$, the focus of the blame-assignment process moves to identify why $v(i_{in})_{actual}$ and not $v(i_{in})_{preferred}$ was produced as the value for i_{in}, instead of identifying why $v(i)_{actual}$ and not $v(i)_{preferred}$ was produced as the value of i (line 3a).

The way that the blame-assignment process infers alternative values for the input of the under-inspection task depends on the type of the failing semantic relation. If the semantic relation violated by the feedback is evaluated by a predicate, the inverse predicate may also be known to the problem solver. In that case, AUTOGNOSTIC applies the inverse predicate to the desired output of the task, i.e., the feedback, and infers the possible alternatives for its input. If the failing semantic relation is a relation exhaustively described in an associative truth table, then the inverse mappings can be inferred from this table. Finally, if the input information is a type of domain object the instances of which belong in some exhaustively described domain, the blame-assignment process can search the domain to find these values which, with the feedback, would satisfy the semantic relations of the task at hand.

Again, it is important to notice here that it is the functional semantics of the task that enable the process of blame-assignment to infer what should have been the "right" input in order for the desired output to have been produced. Furthermore, the rules of composition of the problem-solver's tasks, as captured in the specification of its methods, enable the blame-assignment process to focus the search for the cause of the failure from higher-level tasks to lower-level ones, and from complex types of information (like the solution) to simpler ones on which they depend.

If AUTOGNOSTIC is not able to infer possible alternative values for the input of the task under inspection that can satisfy its semantic relations (line 3b), then it infers that the following causes might be responsible for the failure:

- if the failing semantic relations refer to domain relations, which are exhaustively described by truth tables, AUTOGNOSTIC infers that the content of these relations might be incorrect, and suggests the updating of the domain knowledge to include the mapping of the actual input of the task to its desired output;

- if the failing semantic relations refer to organizational relations, which are exhaustively described by truth tables, AUTOGNOSTIC infers that its knowledge might be incorrectly organized, and suggests the reorganization of the domain knowledge to include the mapping of the actual input to its desired output;

- finally, AUTOGNOSTIC postulates that the reason why the task did not produce the desired output is because it is "incorrectly designed" and its information-transformation function is over-constrained; therefore, it suggests the extension of the task functionality, so that it allows the (currently "illegal") mapping of the actual input of the task to its desired output.

A suggestion for extending a leaf task functionality recognizes essentially that the conceptualization of the role of this task is wrong in the context of the overall task in which it is performed. When the blame-assignment process suggests the extension of the functionality of a leaf task, it has already established that the feedback is within the class of solutions of the overall task. Furthermore, it has established a specific set of values corresponding to the desired input and the output of the task at hand, so that the feedback can be produced, and this set is in conflict with the abstract functional specification of the task. Thus, at this point, the blame-assignment process knows of an input-output transformation, which is desired of the task in the context of this particular problem-solving episode, but which was not intended of the task. Therefore, it postulates the need for re-designing the task in question. If the system did not have the explicit comprehension of the intended irrespective of its actual behavior, this distinction would have been impossible.

The blame-assignment process may reach a leaf task which can produce two alternative values, both of them consistent with its semantic relations, one of which leads to the desired feedback solution and the other to the solution actually produced, which is unacceptable. This is an indication

that the task structure allows multiple solutions to be produced for a given problem, and therefore it is not sufficiently constrained to producing the right kind of solutions. In this case, the following causes for the failure are postulated (line 4):

- if there is a semantic relation of the task, which refers to a domain relation exhaustively described by a truth table, then AUTOGNOSTIC postulates that the content of these relations might be incorrect, and suggests the updating of the domain knowledge to exclude the mapping of the task input to the output it actually produced;
- if there is a semantic relation of the task, which refers to an organizational relation exhaustively described by a truth table, AUTOGNOSTIC infers that its knowledge might be incorrectly organized, and suggests the reorganization of the domain knowledge to exclude the mapping of the task input to the output it actually produced;
- finally, AUTOGNOSTIC postulates that the design of the information-transformation function of the task is under-constrained, and suggests its refinement, in such a way that, the output it actually produced will no longer be acceptable by the new, more constrained, "definition" of the task.

Notice, that the reason that AUTOGNOSTIC suggests the updating (or the reorganization) of the relations to which the task semantics refer, is that it assumes that when the functionality of some task is described in terms of a particular relation it is because the task actually draws its inferences based on that relation. If this assumption is not true, i.e., if a task t does not consult relation rel although its semantics refer to that relation, then modifying the contents of this relation will not affect the actual inferences that the task will draw, and therefore will not prevent the task from drawing the same incorrect inferences as before.

Example Problem 1: AUTOGNOSTICONROUTER is presented with the problem of planning a route from *(10th & center)* to *(ferst-1 & dalney)*, for which it produces the path *((center 10th) (10th atlantic) (atlantic ferst-1) (ferst-1 dalney))*. This path is correct, but suboptimal to the path presented to AUTOGNOSTIC as feedback *((center 10th) (10th dalney) (dalney ferst-1))*.

In ROUTER's model of its navigation world, there are two different neighborhoods which contain both the initial and final intersections of this problem. Having chosen arbitrarily which one among them to search, the problem solver chose the higher-level one. Because at that high-level not much detail regarding the navigation space is available, the problem solver produced a path which consists of major pathways and does not make use of available shortcuts.

In this example, AUTOGNOSTIC's blame-assignment process first focuses on the route-planning task, as the highest task producing the path, and consequently on the path-increase task, as the subtask of intrazonal-method, the method used for route-planning when both intersections belong in the same neighborhood. The semantic relation of path-increase, ForAll n in nodes(path) zone-intersections(initial-zone n), fails for the desired path value because the intersection *10th & dalney)* does not belong in the neighborhood *z1*, the value ROUTER has produced as initial-zone. The relation zone-intersections is an organizational relation, and from its truth table, the alternative initial-zone value is inferred, *za*. Thus, the blame-assignment process focuses on identifying why *za* was not produced as the value for initial-zone. The task producing the initial-zone is the task elaboration. Its semantic relations verify both *za* and *z1* as initial-zone values. Therefore the blame-assignment process suggests as possible modifications the reorganization of the relation zone-intersections so that *(z1 (10th center))* $\notin$ tt(zone-intersections), or alternatively, the refinement the elaboration task information transformation, so that the value it actually produced for the information initial-zone, *z1*, will no longer be legal.

6.6.3 Exploring Alternative Strategies

Often, the desired solution cannot be produced by the same reasoning strategy, i.e., the same task decomposition, that led to the failed solution. Different methods produce solutions of different qualities, and often the desired solution exhibits qualities characteristic of a method other than the one used for problem solving. AUTOGNOSTIC's blame-assignment process recognizes this "incompatibility" between the desired solution and the method used for problem solving, when the OVER-CONSTRAINED-TASK-FUNCTIONALITY (of some task in the strategy used for problem solving) is among the possible causes for the failure. This potential cause and the modification it implies (i.e., the redesign of a task) is evidence that the feedback is in conflict with the very definition of some subtask involved in the task decomposition actually used for problem solving. Before revising the very definition of this subtask (hereafter, this task will be refer to as, the problem task, $t_{problem}$) to resolve the conflict, it is worthwhile investigating whether it is possible to pursue another course of reasoning which will avoid this task, and will produce the feedback solution. Figure 6.6 shows in detail the algorithm for blame assignment within strategies alternative to the one actually used for problem solving.

AUTOGNOSTIC identifies the last task in the task structure before the problem task, for which there exist multiple methods, and which, during problem solving, was accomplished by a method that resulted in the problem task (hereafter, this task will be refer to as choice task, t_{choice}) (line 1). If, during problem solving, at the time of method selection for the choice task, more than one method were applicable, this is an indication that another method should have been chosen (line 3). Therefore, AUTOGNOSTIC proceeds to solve the choice task with each one of the methods applicable to it and not chosen during problem solving (line 4). If one of these methods result in the feedback solution, the blame-assignment process postulates that a possible cause for the failure might be the ill-defined method-selection criteria which allowed the selection of an inappropriate method when there was another one better suited for the problem at hand. This potential cause implies as a possible modification the refinement of the method-selection criteria, so that under similar circumstances the alternative successful method is chosen over the method actually used.

If none of the alternative methods were applicable at the time of method selection, then this is an indication that the problem solver may need to acquire another method (line 5). It is possible, however, that the problem-solver's method selection criteria are wrong, and a particular alternative method could have produced the feedback solution, although during problem solving it was evaluated as not applicable. To collect evidence for that potential, the blame-assignment process evaluates the semantic relations of the tasks arising from the decomposition of the choice task by each alternative method (line 6). If, there is a method, which results in tasks with semantic relations relating the information type of the solution with types of information available at the time of method selection, and if these relations are verified by the feedback, then this is evidence that indeed the feedback solution fits the "quality" of the solutions that this method produces. Therefore, although the method was not evaluated to be applicable at the time, it maybe should have been. Therefore, the blame-assignment process may suggest that the problem solver should try to invoke this alternative method. AUTOGNOSTIC proceeds to solve the choice task with this method, and if the feedback solution is produced, then it postulates the following potential causes for its failure:

- if the method-selection criteria refer to domain relations exhaustively described by a truth table, AUTOGNOSTIC suggests that its domain knowledge might be incorrect and this is the reason the criteria for selecting the rejected method were evaluated to be false; thus, updating the domain knowledge to include the tuple for which the relation was evaluated at the time of method selection, might solve the problem;

- if the method-selection criteria refer to organizational relations exhaustively described by a truth table, AUTOGNOSTIC suggests that its domain knowledge might be incorrectly organized; thus, the reorganization of the domain knowledge to include the tuple for which the relation was evaluated at the time of method selection, might eliminate the cause of the failure;

- finally, AUTOGNOSTIC postulates that the criteria might be incorrect, and therefore, suggests their modification in such a way that the selection criteria of the non-selected but successful

method will be verified by the relevant values during problem solving, and therefore the method will be applicable in similar situations in the future.

It is important to note here that AUTOGNOSTIC understands the different kinds of qualities that different subtasks impose on the types of information they produce in terms of the functional semantics of the tasks which characterize the information transformations they perform. It is this knowledge that enables it, given a desired solution, to infer which reasoning strategy could potentially produce the solution in question, even when the methods involved in that strategy are thought to be inapplicable in the situation.

Example Problem 5: AUTOGNOSTICONROUTER is presented with the problem of going from *(fowler & 3rd)* to *(fowler & 4th)*, for which it produces the path *((fowler 3rd) (3rd techwood) (techwood 5th) (5th fowler) (fowler 4th))*. Although spatially close, the initial and the destination locations in the problem of example 5 belong in different neighborhoods, and thus AUTOGNOSTICONROUTER uses the interzonal-search method to solve the problem. The desired path presented to AUTOGNOSTICONROUTER as feedback is *((fowler 3rd) (fowler 4th))*.

AUTOGNOSTIC while assigning blame for the failure within the strategy used in the problem-solving episode identifies that the feedback path violates the semantic relation of the mbr-path-synthesis subtask of the interzonal-search method. This subtask synthesizes the overall path from smaller paths, produced as solutions to the subproblems into which the original problem is decomposed. It takes as input three paths and concatenates them; as a result, the length of the paths it produces is greater than six nodes, which is not true for the feedback path.

Instead of redefining the mbr-path-synthesis task, in order for the feedback path not to conflict with its current specification (as captured by its functional semantics), AUTOGNOSTIC attempts to identify the cause of the failure to the fact that it did not pursue an alternative method which could have led to the production of the desired path. AUTOGNOSTIC exploring alternative strategies identifies that the semantic relations of path-increase subtask of the intrazonal-search method specify that the paths it produces are shorter than six nodes. This is an indication that intrazonal-search could have produced the path, had it been applicable. Its applicability condition refers to a convention domain relation, zone-intersections(initial-zone final-point). Therefore the blame-assignment process infers that the contents of the convention relation zone-intersections might be incorrect, and thus a potentially appropriate modification would be to include the tuple *(zd (fowler 4th))* in that relation.

6.7 Summary

The goal of the blame-assignment task, in the context of failure-driven learning, is to explain the cause of the problem-solver's failure in terms of some "malfunction" of some of its design elements and thus, set up the context for the repair subtask. One kind of explanation is in terms of the trace of the problem-solver's reasoning on a specific problem. In that view, each particular step is assumed to be correct and only their interactions can be blamed, when they interact in known "pathological" ways. Another kind of explanation is based on the deep comprehension of the functional semantics of the primitive tasks of the problem solver and the rules of their composition into higher-level tasks. Endowing the system with such a model of its own problem solving enables it

- to recognize whether or not the feedback presented to it by the environment is not within its problem-solving competence, without exhaustive problem solving,
- to infer alternatives for particular inferences, relevant to the production of the solution, which could have led to the solution, without having a complete problem-solving trace for the production of the feedback,
- to postulate knowledge conditions which would have allowed these alternative inferences, and

ASSIGN-BLAME-IN-ALT-TS (i,$v(i)_{preferred}$,$t_{problem}$,$trace$,$info\_context$)

Input:
the type of information for which an unacceptable value was produced, i
the feedback value for i, $v(i)_{preferred}$
the task whose functionality has to be extended, $t_{problem}$
the trace of the problem solving, $trace$, and
the information context, $info\_context$.
Output:
a set of possible causes $FailureCauses_{potential}$

---

(1) t_{choice}=$t : t \in trace \wedge t_{problem} \in subtree(t_{choice}, trace) \wedge \exists\ m \in by(t) : t_{problem} \notin subtree(t_i, m)$
(2) $methods_{alt}$=$by(t_{choice}) - \{m\} : m \in trace$
(3) If $\exists\ m_{alt} \in methods_{alt} : applicable(m_{alt}, t_{choice}) \in trace$)
(4) then alt_trace := trace(MONITOR($task_{choice}$, $info\_context$, m_{alt}))
If success
then Include in $FailureCauses_{potential}$
UNDER-CONSTRAINED-METHOD-SELECTION-CRITERIA
to prefer m_{alt} over m
(5) else Include in $FailureCauses_{potential}$
MISSING-METHOD for t_{choice}
(6) $\forall\ m_i \in methods_{alt}$
If $\exists\ rel : rel \in s(t_j) : t_j \in subtree(t_i, m_i) \wedge rel : i \rightarrow i_k \wedge true(rel(v(i)_{preferred}, v(i_k)_{actual}))$
then alt_trace := trace(MONITOR($task_{choice}$, $info\_context$, m_i))
If success
then $\forall\ rel \in c(m_i) : rel : i_k \rightarrow i_j \wedge\ \exists\ tt(rel)$
Include in $FailureCauses_{potential}$
If $indx(rel)$ **then** INCORRECT-KNOWLEDGE-ORGANIZATION
else INCORRECT-DOMAIN-RELATION
Include $(v(i_k)_{actual}\ v(i_j)_{actual})$ in $tt(rel)$
Include in $FailureCauses_{potential}$
OVER-CONSTRAINED-METHOD-SELECTION-CRITERIA
to allow the application of m_i in similar situations

Figure 6.6: The algorithm for Assigning Blame within alternative strategies.

- if such knowledge conditions are not possible, to postulate the need for redesigning the types of inferences its primitive tasks draw, by refining or extending them.

CHAPTER VII

REPAIR AND VERIFICATION

In all failure-driven learning methods, each type of cause of failure that the blame-assignment task is able to recognize corresponds to one or more plans for addressing this particular type of cause. Thus, after identifying the possible cause(s) of the problem-solver's failure, the next step in the learning process is to select one applicable repair plan.

Deciding which particular modification to perform on the failing problem solver is a very hard problem for the following two reasons:

1. In general, if the blame-assignment process does not make the "single-fault assumption", it may identify several potential causes for the problem-solver's failure.

2. Furthermore, there may be several kinds of modifications that can potentially result in the elimination of an error of a particular type.

Thus, after the blame-assignment task has been completed, the learning process has to decide which of the possible causes it has identified it should attempt to repair, and using which particular adaptation.

7.1 Selecting Which Potential Cause of Failure to Address

There are several possible approaches that one can adopt to resolving the first question. For example, the system might decide to eliminate all the possible causes that the blame-assignment task has identified. Lex [Mitchell *et al.* 1981] is a system that adopts this approach. Lex is able to recognize multiple instances of potentially erroneous, operator-selection heuristics in a single problem-solving episode, and during learning it addresses each one of them. Lex admits only one type of error that can possibly exist in the problem-solver's reasoning. Since, and all the causes identified by the blame-assignment task are of the same type, Lex does not have any basis for selecting which causes to address after a failed problem-solving episode.

If, on the other hand, the system is able to recognize several different types of causes of failures, then it may be able to treat the potential errors identified by the blame-assignment task differently, depending on their respective types. AUTOGNOSTIC's method for self-repair exemplifies the latter approach. AUTOGNOSTIC's blame-assignment task is able to recognize two different kinds of potential causes of failure: *domain-knowledge errors* and *task-structure errors*. Of these types of errors it can proceed to repair errors in the content, and the organization of the domain knowledge, and errors of over- or under-constrained task functionalities, and errors in the organization of the task structure. Having a taxonomy of different kinds of causes of failure, AUTOGNOSTIC is able to differentiate among the potential causes identified by the blame-assignment task, and uses the criteria below to select which of the alternatives to address:

- if the failure could be caused by gaps in the problem-solver's knowledge, prefer to address these causes first;

- if the failure could be caused by the under-constrained formulation of the functionality of some tasks, prefer to address these causes next;

- if the failure could be caused by incorrect organization of its domain knowledge, prefer to address these causes next;
- if the failure could be caused by over-constrained formulation of the functionality of some tasks, prefer to address these causes next.

These heuristics do not aim to the least expensive modification but rather to the most "necessary" modification. For instance, errors in the content of the system's domain knowledge are more critical than others, since as long as the problem solver does not have the knowledge relevant to producing the desired solution to a particular problem, it will always fail, irrespective of whether or not the rest of its knowledge is organized correctly and whether or not its task structure is correct. In the same spirit, these heuristics prefer the more general and more expensive modification of refining the functionality of an under-constrained task in the problem-solver's task structure, in order to tailor it to produce the right kind of solutions, instead of simply reorganizing the domain knowledge on which this task depends. The underlying goal is to make the modifications as general as possible, in order to make use of each failed episode as much as possible. Since the formulation of the right class of inferences for a given task will affect the behavior of the problem solver in several problems, where the reorganization of the specific piece of knowledge on which this task depends in the context of the problem at hand will only have an affect ihe current context, AUTOGNOSTIC prefers to attempt the former adaptation first.

7.2 Selecting Which Repair Plan to Use

Even when the system has decided on "which potential cause" of its failure to address, the issue of "how to fix it" still remains open. This is because, in general, there may be several different methods for repairing a given type of cause of failure. To address this issue, some systems employ a single learning method for all the types of failure they can address. Prodigy [Carbonell *et al.* 1989] is a system which employs a single method, explanation-based learning, to repair the errors of its problem solving. This approach is possible when the learning system performs a single learning task, which is the case in Prodigy. The blame-assignment task in Prodigy distinguishes causes of failures in four different types. However, all of them are of the same nature, i.e., erroneous operator-selection, although the context in which each one occurs is slightly different. Thus for all these different types of failure, the goal of the learning task is the same, i.e., to learn operator-selection heuristics.

If, however, the system performs several learning tasks, different in nature, then it must have multiple learning methods to address them. Furthermore, a system may have several alternative methods for the same learning task. AUTOGNOSTIC adopts this latter approach. For some types of causes of failures, AUTOGNOSTIC knows multiple repair plans that it can potentially use to eliminate them. When the blame-assignment process has identified an instance of one of these types, if AUTOGNOSTIC decides to address it, it decides on which particular method to use, by evaluating which is the least expensive one, under the particular knowledge conditions of the situation. Table 7.1 illustrates AUTOGNOSTIC's taxonomy of types of causes of failure, and the plans which it can use to eliminate them.

For two of the types of causes of failure that AUTOGNOSTIC can address, namely the under- and over-constrained task functionality, it has several repair plans which it can use to do so. To select among them, AUTOGNOSTIC evaluates the knowledge conditions of the situation with respect to the knowledge requirements of each particular plan and based on the relative cost of each plan. Thus, in both cases, the substitution of the failing task with another is the less costly repair plan, however it assumes that there exist tasks similar (i.e., instances of the same prototype task) to the failing one. If this is true in the particular situation, AUTOGNOSTIC prefers the task-substitution plan. Otherwise, in the case of over-constrained task functionality, it has to modify the failing task. In the case of under-constrained task functionality, AUTOGNOSTIC has still to decide between the task-modification and the selection-task-introduction plans. Both these plans are of similar cost, since they both require the discovery of a new semantic relation to constrain the functionality of the failing task. AUTOGNOSTIC prefers the task-modification plan when the failing task produces a single type

Table 7.1: Repair: From Causes of Failure to Repair Plans.

	Causes of Failure	**Repair Plans**
Domain Knowledge Modifications	Incorrect Object Representation	Plan for Revising an Object Representation *not implemented*
	Unknown Object	Plan for KA of new Object
	Incorrect Domain Relation	Plan for Updating an Enumerated Relation
	Incorrect Knowledge Organization	Plan for Updating an Enumerated Relation
Task Structure Modifications	Incorrect TS Organization (MI)	Plan for TS Reorganization
	Incorrect TS Organization (VS)	Plan for TS Reorganization
	Under-Constrained Task functionality	Plan for Task Substitution Plan for Selection-Task Introduction Plan for Task Modification
	Over-Constrained Task functionality	Plan for Task Substitution Plan for Task Modification
	Task Semantics and Process Mismatch	Plan for Task-Semantics Revision
	Missing Method	Plan for Method Acquisition *not implemented*
	Under-Constrained Method-Selection Criteria	Plan for Method-Selection Criteria Refinement *not implemented*
	Over-Constrained Method-Selection Criteria	Plan for Method-Selection Criteria Extension *not implemented*

of information as output, where it prefers the selection-task-introduction plan otherwise. This heuristic aims to simplify the re-programming task of the repair. If the failing task produces several outputs, and its functionality with respect to one of them needs to be modified, then modifying the procedure carrying out this task might potentially disturb its functionality with respect to the rest of its outputs. Thus, in this cases AUTOGNOSTIC prefers to simply refine the functionality of the failing task with a new task selecting the particular output of interest, namely the output for which the functionality of the task is under-constrained.

7.3 Maintaining the Integrity of the Problem Solver

Once the system has selected a specific plan that can eliminate the cause of its failure, then it can proceed to apply it. When the invocation of the chosen plan involves the modification of the problem-solving process, a new issue, critical to the overall success of the failure-driven learning method, arises:

> How can the system modify its own problem-solving process in a way that maintains its overall consistency and integrity?

This issue does not arise when the system does not have the ability to modify its set of primitive reasoning steps, the primitive functional elements it is designed with. In Lex or Prodigy for example, the consistency of the problem-solving process cannot be compromised by new heuristics for selecting among operators, because these pieces of knowledge exactly match roles which are pre-defined by the problem-solving process. However, substituting a task with a new one, or introducing a new task altogether, has more drastic affects to the problem-solving process. There are no pre-defined roles for the new task, that is, its information and control interactions with the rest of the task structure are still to be defined. In AUTOGNOSTIC, the repair plans which modify the problem-solver's task structure, are guided by AUTOGNOSTIC's comprehension of the information and control interactions of the problem-solver's current tasks and methods, as captured in the SBF model of the problem solver.

> The SBF model of the problem solver enables AUTOGNOSTIC
>
> - to recognize tasks whose information-transformation function could potentially substitute failing tasks of the problem solver,
> - to postulate new information-transformation functionalities for new tasks which are missing from the task structure, and
> - to integrate the new tasks within the control flow of the problem-solver's current task structure.

Once the repair task is completed, AUTOGNOSTIC evaluates its success by attempting to solve the problem that caused the failure in response to which the modification was performed. If the problem is successfully solved, the repair is evaluated to be correct. Otherwise, AUTOGNOSTIC proceeds to apply an alternative repair plan, if multiple repairs plans are applicable to the cause of failure it is addressing. This, for example, happens in the case of the task-modification repair plan, when AUTOGNOSTIC has discovered multiple relations that can be used to further constrain (or expand) the functionality of a failing task. In such situations, AUTOGNOSTIC may instantiate the task-modification repair plan with several of these potential relations until one is found to be successful. If there are no other repairs that it can invoke to eliminate the cause of failure it is currently addressing, AUTOGNOSTIC attempts to eliminate other causes that the blame-assignment task has potentially identified. If there are no alternative causes, it proceeds to a new blame-assignment and repair cycle.

7.4 Domain-Knowledge Modifications

7.4.1 Acquiring Knowledge about a New Domain Object

AUTOGNOSTIC interacts with its environment by exchanging information with it. The environment presents AUTOGNOSTIC with problems and AUTOGNOSTIC returns to the environment the solutions it produces. Sometimes, the environment may evaluate AUTOGNOSTIC's solutions to be unsatisfactory, and then it provides some feedback to AUTOGNOSTIC to inform it about a preferable solution. At any point where the environment presents AUTOGNOSTIC with some type of information, that is, at the time of presenting a problem along with the information context in which the problem should be solved, and also, at the time of presenting a more satisfactory solution to a particular problem, there is the possibility that this information contains references to objects and relations in the domain that AUTOGNOSTIC does not know about. When AUTOGNOSTIC recognizes that it does not know about a specific object to which the information given by the environment refers, it postulates that its ignorance might potentially be the cause of its failure. To address this type of cause, AUTOGNOSTIC invokes the plan for acquisition of knowledge about a new object.

The *plan for knowledge-acquisition of a new object*, the algorithm for which is shown in Figure 7.1, consists of two subtasks:

1. to include the new object in the problem-solver's domain for this object type, (line 1), and
2. to update the problem-solver's domain relations which refer to this type of object (line 2).

The first subtask is applicable when the unknown instance belongs in an enumerated type of object. The second one is applicable when there are relations which apply to this type of object and whose truth value is exhaustively enumerated in a truth table; these relations need to be informed with entries relating the new instance to the other known instances. Furthermore, if there are higher-level constraints on the updated relations, connecting them to other relations, these other affected relations also need to be updated (line 3). To date, AUTOGNOSTIC simply asks an oracle in order to introduce new entries associated with the new instance, in these relations, but there is no assumption that the oracle will give AUTOGNOSTIC complete or correct information.

Example Problem 4: Let us now discuss the modifications suggested to ROUTER's domain knowledge in the problem of example 4. In this problem, AUTOGNOSTICONROUTER was presented with the problem of going from *(8th-1 & mcmillan)* to *(north & cherry-3)* and produced the path *((8th-1 mcmillan) (mcmillan test-3) (test-3 hemphill) (hemphill ferst-2) (ferst-2 ferst-3) (ferst-3 cherry-3) (cherry-3 north))*. At the end of its problem solving, AUTOGNOSTICONROUTER was presented with an alternative solution, preferable to the one that it had actually produced, i.e., *((8th-1 mcmillan) (mcmillan 6th-1) (6th-1 ferst-2) (ferst-2 ferst-3) (ferst-3 cherry-3) (cherry-3 north))*. The blame-assignment process, in this example, had suggested that the reason why ROUTER was not able to produce the desired path, was because it did not know about the existence of the street *6th-1*, and its intersections *(mcmillan 6th-1)* and *(6th-1 ferst-2)*.

In response to that suggestion, AUTOGNOSTIC first updates the domains of `street` and `intersection` with the new instances *6th-1*, and *(mcmillan 6th-1)* and *(6th-1 ferst-2)* respectively. Subsequently, it proceeds to update the domain relations applicable to them. The only relation in ROUTER's domain theory applicable to streets is the `legal-direction` relation which maps a street to the legal directions of traffic on this street. In its current implementation, in order to acquire the value `legal-direction(` *6th-1* `)` AUTOGNOSTIC has to ask an oracle about it, which informs AUTOGNOSTICONROUTER that `legal-direction(` *6th-1* `)` = *East West*.

In the case of acquiring knowledge about a new intersection, after having added the new entry in the `intersection-domain`, AUTOGNOSTIC has to update the domain relation `zones-of-int` with entries for the two new instances. In its current implementation, the oracle's answer to the `zones-of-int(x?)` is always the same, i.e., *z1*. Essentially, by default all the new intersections that AUTOGNOSTICONROUTER integrates in ROUTER's domain knowledge are added at the level of the highest neighborhood. However, there is a high-level constraint imposed on this

INCLUDE-INSTANCE-IN-DOMAIN(wo, v)

Input:
the type of domain object of which a new instance must be acquired, wo, and
the new instance, v.

Output:
the domain knowledge modified to include knowledge of the new instance.

---

(1) If $domain(wo)$
then Add v in $domain(wo)$

(2) $\forall rel \in relations(wo)$
If $tt(rel)$
then ask oracle about $rel(v)$

(3) propagate-the-constraints(+ rel entry(v))

Figure 7.1: Acquiring Knowledge of a new instance of a domain object.

relation, namely that for each entry $(ab) \in$ zones-of-int , $(ba) \in$ zone-intersections relation. Thus AUTOGNOSTIC propagating the constraints of the addition of *((mcmillan 6th-1) z1)* and *((6th-1 ferst-2) z1)* to the zones-of-int relation, adds in the entry zone-intersections(*z1*) the two new intersections.

7.4.2 Updating an Enumerated Relation

Often, in the process of assigning blame for a particular failure, AUTOGNOSTIC notices that the problem solver did not infer a desired value for a particular type of information because its domain knowledge contains a "fact" conflicting with the desired inference (or alternatively, a fact leading to the inappropriate one) or because is is organized in such a way that does not allow relevant information to be used. AUTOGNOSTIC's ability to recognize such types of potential causes of failure relies in its meta-knowledge about the kinds of domain knowledge that the problem solver uses for each of its tasks. AUTOGNOSTIC's ability to repair them relies on its knowledge about how these types of knowledge are actually represented by the problem solver.

Remember, the SBF specification for domain relations (both state-of-the-world relations and convention relations) includes a description of whether the domain relation is exhaustively enumerated in a truth table, $tt(rel)$, or whether it is generated by a closed-form predicate, $p(rel)$. In the former case, AUTOGNOSTIC can modify this domain relation by updating the contents of its truth table. In the latter case, to modify the domain relation, AUTOGNOSTIC should have knowledge of the semantics of the language in which the problem solver is implemented, in order to modify the predicate generating it, and this knowledge is beyond AUTOGNOSTIC's current status.

Whenever AUTOGNOSTIC notices a potential error in a relation which is implemented in a truth table, it invokes the *plan for updating an enumerated relation*. In principle, however, the system could differentiate between state-of-the-world and convention relations, and it could require additional evidence in order to perform a modification to a relation that is grounded in the state of its external environment. Furthermore, the validity of such a modification could also be evaluated through perception, i.e., the problem solver may explore its environment in order to verify whether its internal representation is faithful to the reality. In the case of convention relations, however, the system may justify its modification solely on the basis of whether or not it enables it to produce

the desired solution for the current problem, i.e., the problem that caused the failure in response to which the modification is proposed.

AUTOGNOSTIC recognizes the need to modify the value of a domain relation for a specific set of values when this domain relation is referred to by a semantic relation (or a condition), and there is evidence that the feedback values are not produced by the problem-solving process, either because the semantic relation (or condition) in question is not true for these values, or because it is true for both the feedback values and the actual ones. In the former case, AUTOGNOSTIC suggests that the value of the domain relation referred to by the semantic relation (or condition) should be made true for the feedback values; that is, the entry consisting of these particular values should be added to the truth table of the relation. In the latter case, it suggests that it should be made false for the actual ones; that is, the entry consisting of the actual values used in problem solving should be deleted from its truth table.

Figures 7.2 and 7.3 depict the algorithms for modifying the contents of the truth table of a domain relation (a modification resulting in acquisition of domain knowledge or reorganization of the domain knowledge, depending on whether the relation is a state-of-the-world relation or a convention relation respectively). These algorithms are symmetrical, and they consist of four basic steps:

1. AUTOGNOSTIC identifies the actual table and the actual entry or entries to be added to (deleted from) the truth table, (line 1),
2. simplifies the expression in the function of the semantic relation (or condition), (lines 2),
3. adds (deletes) these entries to(from) the truth table, (line 3), and
4. propagates the constraints of this modification to the other domain relations which are dependent upon the modified relation, (line 4).

The semantic relations (and conditions) in the SBF model of a problem solver may modify the domain relation, they refer to, in the following ways: *negation* (e.g., NOT(i1 domain-relation o1)), *inversion* (e.g., i1 INV(domain-relation) o1), *universal quantification* (e.g., FORALL i IN i1, i domain-relation o1), and *existential quantification* (e.g., EXISTS i IN i1, i domain-relation o1). Thus, when the blame-assignment process suggests that a particular entry should be added to, or deleted from the domain relation AUTOGNOSTIC has to transform the entry according to qualifications of the semantic relation (or condition) before actually modifying the table of the domain relation it refers to. Next, AUTOGNOSTIC proceeds with the actual modification of the truth table and the propagation of the domain constraints which apply to the modified domain relation.

Example Problem 5: In the problem of example 5, AUTOGNOSTICONROUTER was presented with the problem of going from *(fowler & 3rd)* to *(fowler & 4th)*, for which it produced the path *((fowler 3rd) (3rd techwood) (techwood 5th) (5th fowler) (fowler 4th))*. The environment presented to AUTOGNOSTICONROUTER as feedback an alternative path, *((fowler 3rd) (fowler 4th))*. For this failure, the blame-assignment process identified as a possible cause for the failure, the incorrect organization of the zones-of-int relation.

To address this error, AUTOGNOSTIC's repair process proposes to update the relation in question and to make the relation zones-of-int true for the values *((fowler 4th) zd)*. After having added *zd* to the list of zones in which the intersection *(fowler 4th)* belongs, AUTOGNOSTICONROUTER notices that there is a higher-level constraint among the zone-intersections and zones-of-int relations. Namely, if there is an entry *zones-of-int(int? z?)* then there should also be an entry *zone-intersections(z? int?)*. Therefore, AUTOGNOSTICONROUTER also adds the intersection *(fowler 4th)* to the list of intersections that the neighborhood *zd* contains. After this modification, when AUTOGNOSTICONROUTER attempts to solve again this problem, it finds (i.e., from the classification subtask) that the two intersections given as initial and destination location belong in the same neighborhood (where in the previous failed episode they were not). Therefore, where it has used the interzonal search me before, now it proceeds to solve the problem with an intrazonal search within this common neighborhood. This time, AUTOGNOSTICONROUTER returns the desired solution.

Input:
the semantic relation which needs to be modified, rel, and
the tuple $(v(i1), v(i2))$ for which it must be modified.
Output:
the domain knowledge modified.

MAKE-RELATION-TRUE$(rel, v(i1), v(i2))$

(1) $table$:= $tt(function(rel))$
(2) If $negative(function(rel))$
then make-relation-false($not(rel)\ v(i1), v(i2)$)
If $inverse(function(rel))$
then make-relation-true($reverse(rel), v(i2), v(i1)$)
If $universallyquantified(rel)$
then If $quantifiedoninfo1(rel)$
then $V(i1) = f(attr1(rel)\ v(i1))$
loop $v \in V(i1)$ make-relation-true($(rel), v, v(i2)$)
else $V(i2) = f(attr2(rel)\ v(i2))$
loop $v \in V(i2)$ make-relation-true($(rel), v(i1), v$)
(3) If $simple(function(rel))$
then add $(v(i1), v(i2))$ to $table$
(4) propagate-the-constraints(+ function(rel) $(v(i1), v(i2))$)

Figure 7.2: Updating an Enumerated Relation: Making a relation true for a given pair of values.

In this problem, AUTOGNOSTICONROUTER's learning resulted in reorganization of its domain knowledge, which, in turn, enabled ROUTER to use a method (i.e., intrazonal-search method) which was previously inapplicable. AUTOGNOSTIC's capability to modify the organization of its domain knowledge in order to enable the use of some methods vs. others has very significant implications. Essentially, it means that a problem solver may begin with a problem-solving process which can make use of associations between certain concept types, but without any preconceived notions of what exactly associations it should have. The exact associations can be developed on demand, through AUTOGNOSTIC's reflective learning process. AUTOGNOSTIC is able to identify when a particular association is necessary in order for the problem-solving process to produce the kinds of solutions that are desired from the environment. Thus, instead of committing to particular associations at the time of designing the problem solver, which may require considerable analysis, the designer need only to identify the "nature" (i.e., which domain objects should be related) of the associations that the problem-solving process depends upon, and to make this nature explicit in the SBF model of the problem solver.

The reflective domain-knowledge organization is intuitively more plausible from its alternative, also when evaluated in terms of its cognitive plausibility. The alternative approach requires the initialization of the associations at some specific point in time, presumably before the problem solver first starts to solve problems, and it is difficult to imagine on the what basis this initialization

Input:
the semantic relation which needs to be modified, rel, and
the tuple $(v(i1), v(i2))$ for which it must be modified.
Output:
the domain knowledge modified.

MAKE-RELATION-FALSE($rel, v(i1), v(i2)$)

(1) $table$:= $tt(function(rel))$
(2) If $negative(function(rel))$
then make-relation-true($not(rel)\ v(i1), v(i2)$)
If $inverse(function(rel))$
then make-relation-false($reverse(rel), v(i2), v(i1)$)
If $universallyquantified(rel)$
then If $quantifiedoninfo1(rel)$
then $V(i1) = f(attr1(rel)\ v(i1))$
loop $v \in V(i1)$ make-relation-false($(rel), v, v(i2)$)
else $V(i2) = f(attr2(rel)\ v(i2))$
loop $v \in V(i2)$ make-relation-false($(rel), v(i1), v$)
(3) If $simple(function(rel))$
then add $(v(i1), v(i2))$ to $domain(function(rel))$
(4) propagate-the-constraints(- function(rel) $(v(i1), v(i2))$)

Figure 7.3: Updating an Enumerated Relation: Making a relation false for a given pair of values.

would be made. One possibility would be through interaction with other systems who have already been solving problems in the same domain (i.e., social intelligence). That is, the novice system can ask the experts which exactly are the particular associations it needs, and it could literally initialize its domain-knowledge organization with these associations. Although, this is not too unrealistic, there is ample psychological evidence that this kind of rote learning is not very effective, as opposed to learning through personal experience.

ROUTER's domain knowledge is organized through three different such relations: zone-intersections, zones-of-int, and children-zones. These three relations essentially define the hierarchy of neighborhoods in ROUTER's world model. An extensive experiment was conducted [Stroulia and Goel 1994a] to investigate AUTOGNOSTIC's capability to induce these relations on an initially "flat" world model. In this experiment, ROUTER was initialized with only one neighborhood, *z1*, which contained all known intersections. The initial status of these domain relations of ROUTER's is shown in Figure 7.4.

The following example illustrates how these "empty" convention domain relations were filled in through AUTOGNOSTIC's reflective learning.

zone-intersections
{ *(z1 ((10th northside) (10th hemphill) ...))* }
zones-of-int
{ *((10th northside) (z1)) ((10th hemphill) (z1)) ...* }
children-zones
{ *(z1 (z1))* }

Figure 7.4: The initial content of the convention domain relations, zone-intersections, zones-of-int, and children-zones, in ROUTER.

Example Problem 6: AUTOGNOSTICONROUTER is presented with the problem of going from *(10th & center)* to *(5th & techwood)*, for which it produces the path *((10th center) (10th dalney) (10th atlantic) (10th fowler) (10th techwood) (techwood 8th-3) (techwood 6th-2) (techwood 5th)).*

Because at this point AUTOGNOSTICONROUTER does not have a hierarchy of neighborhoods yet, it solves this problem through an intrazonal search, since all the intersections it knows about belong in the same neighborhood. As it monitors its progress, it recognizes that the path it has come up with is longer than the length allowed by the semantics of the intrazonal-search task. The requirements, imposed by the semantics of the tasks intrazonal-search and mbr-path-synthesis on the length of the produced path, assume the existence of the hierarchical organization on ROUTER's domain knowledge. The nature of this organization is the reason why intrazonal-search produces in general short paths, where mbr-path-synthesis produces longer ones. In other words, it is because neighborhoods organize intersections in small localities, that search within a neighborhood results in short paths, where the combination of paths produced through search in several neighborhoods tend to be long. Since however, the actual contents of these relations are not in congruence with the nature of the hierarchy (because they are defaults) the semantics of these task fail.

In this example, the blame-assignment process suggests that AUTOGNOSTICONROUTER should make the relation zone-intersections false for the values *(z1 (techwood 5th))*. Having deleted *((techwood 5th) z1)* from the set of intersections which belong in the neighborhood *z1*, AUTOGNOSTICONROUTER propagates the affects of this modifications to the relation zones-of-int and deletes *z1* from the neighborhoods that the intersection in question belongs to. AUTOGNOSTICONROUTER realizes that this intersection does not belong to any neighborhood anymore, which is not acceptable. To remedy the situation, AUTOGNOSTICONROUTER postulates the existence of another neighborhood in which this intersection should belong from now on. Thus, it spawns a process for acquiring a new instance of a domain concept, to introduce a new neighborhood in its domain knowledge.

This is essentially the same process which is invoked when AUTOGNOSTIC realizes that its external agent refers to an unknown instance of a domain concept, in its problem specification or in its feedback information, shown in Figure 7.1. The oracle, which answers AUTOGNOSTICONROUTER's questions regarding the domain relations applicable to neighborhoods, tells AUTOGNOSTICONROUTER that the new neighborhood should belong in children-zones of the old one, and it should contain as its zone-intersections a tree of intersections around the one which caused the introduction of the new neighborhood in the first place. Through propagation of these modifications, the zones-of-int relation is also modified to reflect that the intersections in the zone-intersections(*new-zone*) belong in that neighborhood.

7.5 Task-Structure Modifications

7.5.1 Reorganizing the Task Structure

AUTOGNOSTIC reorganizes its task structure when a method, evaluated as applicable to a task in the context of a particular problem-solving episode, fails to complete the task in question. There are two types of failures which can prevent a method from accomplishing a task:

1. When a subtask of the method cannot be accomplished because it consumes as input some type of information which should have been produced by an earlier subtask of the method, but which is not available in the particular episode.

   In that case, AUTOGNOSTIC removes the subtask, which should have produced but failed to produce the information in question, from the set of the method's subtasks. Furthermore, it adds to the list of the method's applicability conditions, the existence of a legal value for the type of information in question.

2. When a subtask of the method produces for its output values that fail to meet its semantics.

   In that case, AUTOGNOSTIC again removes the subtask from the set of the method's subtasks, and it adds to the list of the method's applicability conditions the verification of the semantics of this subtask by its output.

When, during a particular problem-solving episode, a method fails to accomplish the task it is applied to, although its applicability conditions are met, AUTOGNOSTIC recognizes an opportunity for improving its understanding of the situations in which the method in question is indeed useful. When the cause of the failure is internal to the method, i.e., there are pathological information and control interactions among the subtasks of the method, the following paradox may occur: the revision of the method conditions (to include the availability of the missing information, or the verification of the violated semantics) introduces a condition which assumes the existence of some type of information (the information currently missing or the output information which currently violates the semantics) which is not produced by some earlier task, and therefore is not available at the time of the condition evaluation. This phenomenon violates the semantics of the SBF language, and in turn gives rise to a new modification, i.e., the reorganization of the task structure to move the production of the information in question before the evaluation of the method conditions. Therefore, AUTOGNOSTIC proceeds to reorganize its task structure in order to make these interactions explicit. Both the above modifications, i.e., the revision of the method-selection criteria, and the reorganization of the task structure, are the two subtasks of the *plan for task reorganization*. Essentially, AUTOGNOSTIC recognizes that in order to make the employment of the problem-solver's methods more effective, it has to make more informed decisions; and in order to make more informed decisions, it has to have all the information pertinent to its decision available to it at this point in its control of processing when it is actually making the decision. Thus it recognizes an information dependency among a task (the one responsible for producing the missing information, or the one whose semantics have failed) and a method (the one whose applicability is conditioned upon the existence of the information or the validation of the semantics respectively) which necessitates a modification to the control flow in the task structure; namely, it necessitates the performance of the subtask before the method of the selection.

The algorithm for the actual task reorganization is shown in Figure 7.6. To move the task producing the information upon which the selection of the method m is conditioned, $t_{producer}$, AUTOGNOSTIC introduces a new level in its task structure. Essentially, it introduces a new task t', (line 2), and new method m' (line 3) between the task t_{failed} to which the method m is applicable. In the modified task structure, the task t_{failed} is going to be accomplished by a single method m' (line 4) which will set up two tasks: $t_{producer}$ and t' which will in turn be accomplished by the methods that used to be applicable to t_{failed}, m being one among them. Figure 7.5 illustrates the differences between the task structure before and after this modification.

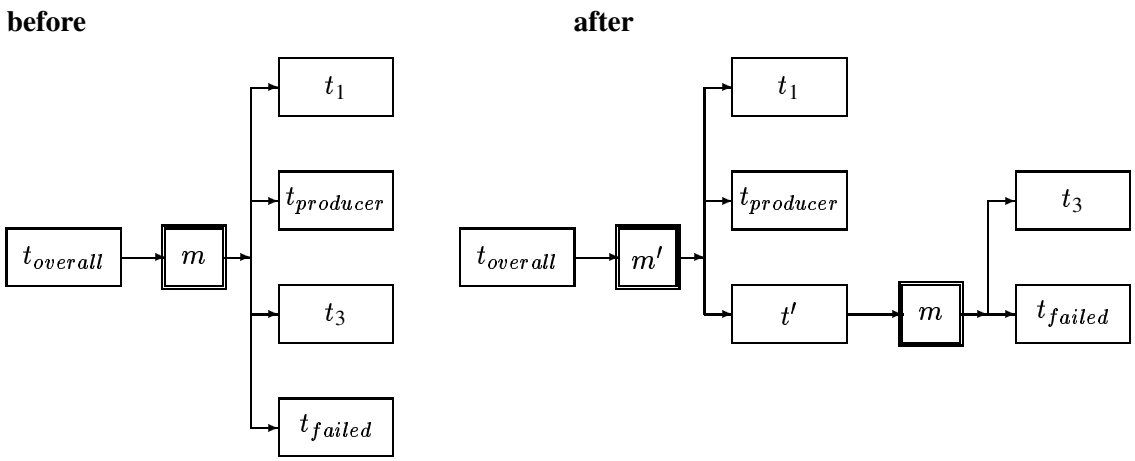


Figure 7.5: Task Structure before and after reorganization.

TASK-REORGANIZATION$(m, info, t_{producer})$

Input:
the failed method, m
the type of information whose value is missing (or which violates the semantics), $info$, and
the method's subtask producing $info$, $t_{producer}$

Output:
a reorganized task structure.

---

(1) Let t_{failed} be the task in service of which, method m was invoked

(2) Create a new task t', such that

$by(t') := by(t_{failed})$

$i(t') := i(t_{failed}) \cup \{info\}$

$o(t') := o(t_{failed})$

(3) Create a new method m', such that

$subtasks(m') := \{t_{producer}, t'\}$

$conditions(m') := \emptyset$

(4) $by(t_{failed}) := \{m'\}$

Figure 7.6: Reorganizing the task structure to bring the performance of a task before the evaluation of a method's applicability.

Example Problem 3: In the problem of example 3, AUTOGNOSTICONKRITIK2 was presented with the problem of designing a nitric acid cooler that cools nitric acid from temperature $T1$ to temperature $T2$, where $T2 << T1$. It retrieved from its design memory a nitric acid cooler which

performs a variant of the currently desired function, namely it cools nitric acid from temperature $T1$ to temperature $T2_{old}$ where $T2_{old} << T1$ but $T2_{old} > T2$. While attempting to adapt the old design by substituting its water pump with a new one, AUTOGNOSTICONKRITIK2 noticed that it did not have an appropriate alternative water pump in its component memory, and therefore it could not successfully complete the component replacement method. The subsequent blame-assignment process suggested the addition of a new condition to the applicability of KRITIK2's component-replacement method, i.e., the availability of a replacement which it can use to replace the problematic component in the old device (in the example *water_pump*). This modification is very simple; it only involves the addition of the relation not(null(replacement)) in the under-condition slot of the frame describing this method. In order however to make the evaluation of this new condition possible at the time of method selection, AUTOGNOSTICONKRITIK2 has also to reorganize KRITIK2's task structure to bring forward the subtask which is responsible for producing the replacement . The reorganization of KRITIK2's task structure, caused by the example problem 3, has the affects shown in Figure 7.7. This modification makes explicit a condition for the successful application of the component-replacement method which was implicit in the previous one. As a result, it improves AUTOGNOSTICONKRITIK2's comprehension of the information interdependencies between its tasks, and enables it to use its methods more efficiently.

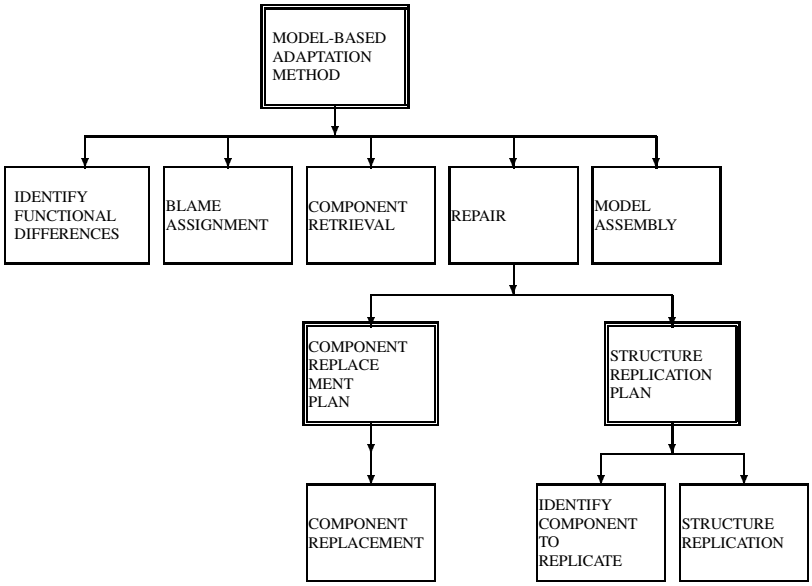


Figure 7.7: The modified task structure of KRITIK2.

7.5.2 Substituting one Task with Another

When the cause of the problem-solver's failure is the over-constrained (or under-constrained) functionality of a particular task in its task structure, a potentially applicable repair plan is the *plan for task substitution*. The goal of this repair plan is to replace the failing task with a new one which produces the same kinds of information as output, potentially with the same types of information as input. However, the mapping between the two should be slightly different than the mapping of the original task, so that the replacement task exhibits the desired behavior in the context of the problem-solving episode in which the to-be-replaced task failed.

In order to substitute a particular task of the problem solver with another, AUTOGNOSTIC has first to identify a potential replacement task, which delivers a variant functionality of the task to be replaced. The SBF language, provides the system with the ability to specify the information-transformation functionality of its tasks. In addition, through the *prototype - instance* relation between tasks, it also enables the system to organize its tasks in a classification hierarchy along their functionalities, to compare the functionalities of two different tasks, and to evaluate their inter-changeability in the context of a particular task structure. Notice that, if the system is not able to characterize its tasks independently from the task structure in which they are employed, then it does not have any basis for comparing two given tasks, unless it already knows that they can play the same role in the same task structure. Therefore, it is unable to modify its task-structure, and consequently its problem-solving process, in any significant way.

When AUTOGNOSTIC invokes the task-substitution plan, it identifies all the different instantiations of the prototype of the task to be replaced. Next, it proceeds to employ these alternative instantiations in the context of the failed problem-solving episode. Essentially, it monitors the behavior of the alternative instantiations, given the input of the task to be replaced in this episode. If there is one instantiation which produces the desired output, then AUTOGNOSTIC proceeds to replace the failed task with this successful instantiation.

Example Problem 7: Let us assume a scenario where ROUTER's procedure for the retrieval task retrieves only paths which match the initial-point of the current problem.[1] An let us also assume that among other paths, ROUTER's memory contains the path *((10th center) (10th atlantic) (home-park atlantic))*. In this knowledge context AUTOGNOSTICONROUTER is presented with the problem of going from *(10th & curran)* to *(peachtree-pl & atlantic)*. For this problem, since the two intersections do not belong in the same neighborhood together, and since no path is retrieved from memory, the planning task is accomplished by the interzonal-search method. The application of this method results in the production of the path *((10th curran) ... (hemphill ferst-1 ferst-2)... (10th atlantic) ... (peachtree-pl atlantic))*.

AUTOGNOSTIC is given as feedback the path *((10th curran) (10th test-3) (10th center) (10th atlantic) (home-park atlantic))*, and it then proceeds to assign blame for its failure to produce the desired path. The feedback path conforms with the semantics of the route-planning and the planning tasks. From its evaluation of the feedback path against the semantics of the dor-path-synthesis task, AUTOGNOSTIC infers that in order for the desired path to have been produced, the middle-path should have been *((10th center) (10th atlantic) (home-park atlantic))*. Thus, AUTOGNOSTIC proceeds to investigate why this value was not produced.

In the process of assigning blame for the non-production of the desired value for the middle-path, AUTOGNOSTIC notices that this particular path could not have been produced by either the middle-subproblem task or the retrieval task since it does not meet their semantics (i.e., not all its intersections belong in the same neighborhood, and its initial node is not the same as the initial location). Thus, is identifies as one potential cause for the failure, the over-constrained information transformation of the retrieval task, and it proceeds to substitute it with another one.

As shown in Appendix 1, AUTOGNOSTICONROUTER knows of several instantiations of the retrieval-prototype task, one of which, retrieval-prime1, is actually used in the current working task structure of ROUTER. AUTOGNOSTICONROUTER invokes each one of the alternatives, one after the other, until one exhibits the desired behavior, i.e., until one produces the desired middle-path. In this example, both alternatives would produce the desired middle-path, but AUTOGNOSTIC stops its search after the first successful candidate, which in this case is retrieval-prime1. At this point, it replaces the retrieval task with a new one, which is an instance of the successful candidate, retrieval-prime1. Figure 7.8 depicts the original task structure of ROUTER (only the high-level decomposition of route-planning) and Figure 7.9 depicts it after the substitution of the retrieval task.

[1] Notice, this is a different set-up than the one described in the example problem 2.

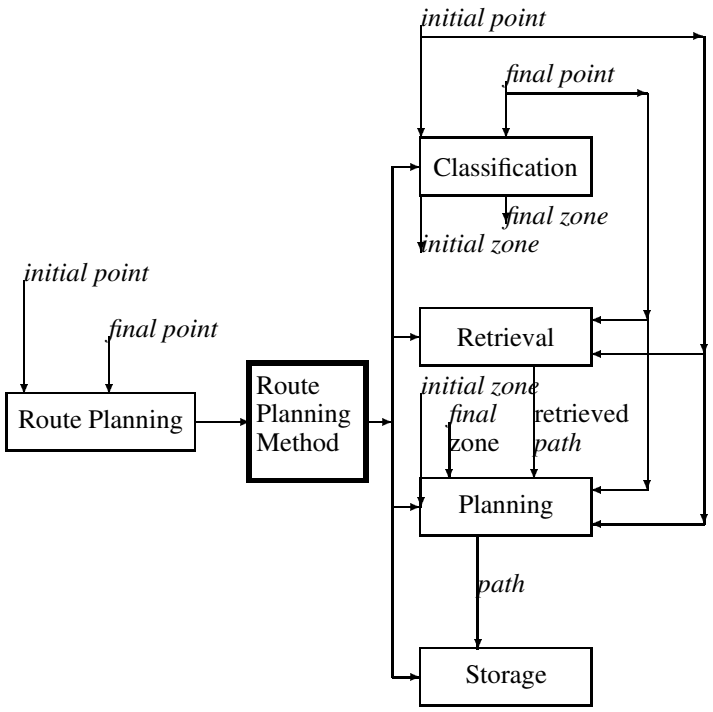


Figure 7.8: The original task structure of ROUTER.

7.5.3 Modifying a Task

When the cause of the problem-solver's failure is the over-constrained (or under-constrained) functionality of a particular task in its task structure, another potentially applicable repair plan is the *plan of task modification*. The goal of this repair plan is to modify the task in question so that its functionality is extended (or refined). In other words, the plan generalizes (or specializes) the class of the values that this task produces for some of its output information, in order to include in this extended class the value desired for this type of information (or exclude from the refined class the value it actually produced).

In order to modify the functionality of a particular task of the problem solver, AUTOGNOSTIC has first to identify which the desired task functionality should be. Essentially, it has to postulate a new abstract functionality for the task to be modified. From the blame-assignment task, AUTOGNOSTIC knows what the desired output of the task should have been in this episode, therefore it already has a specific instantiation of the desired functionality in the context of the failed problem-solving episode. In the case of specializing the functionality of the task, this is equivalent to identifying a criterion which would differentiate the values in the specialized class and the values in the current class of outputs. In the case of generalizing the functionality of the task, it is equivalent to identifying a relation extending the current class of the output of the task to include the additional outputs which are desired of the task.

AUTOGNOSTIC's process for discovering a new relation with which to characterize the new functionality of a task is based on its abstract comprehension of the types of knowledge available to the problem solver. The SBF model of the problem solver explicitly specifies the ontology of the problem-solver's domain. For each type of information that its tasks consume and produce, the SBF model specifies what type of domain concept it is. Moreover, for each type of domain concept, among other things, the model specifies the domain relations which are applicable to it.

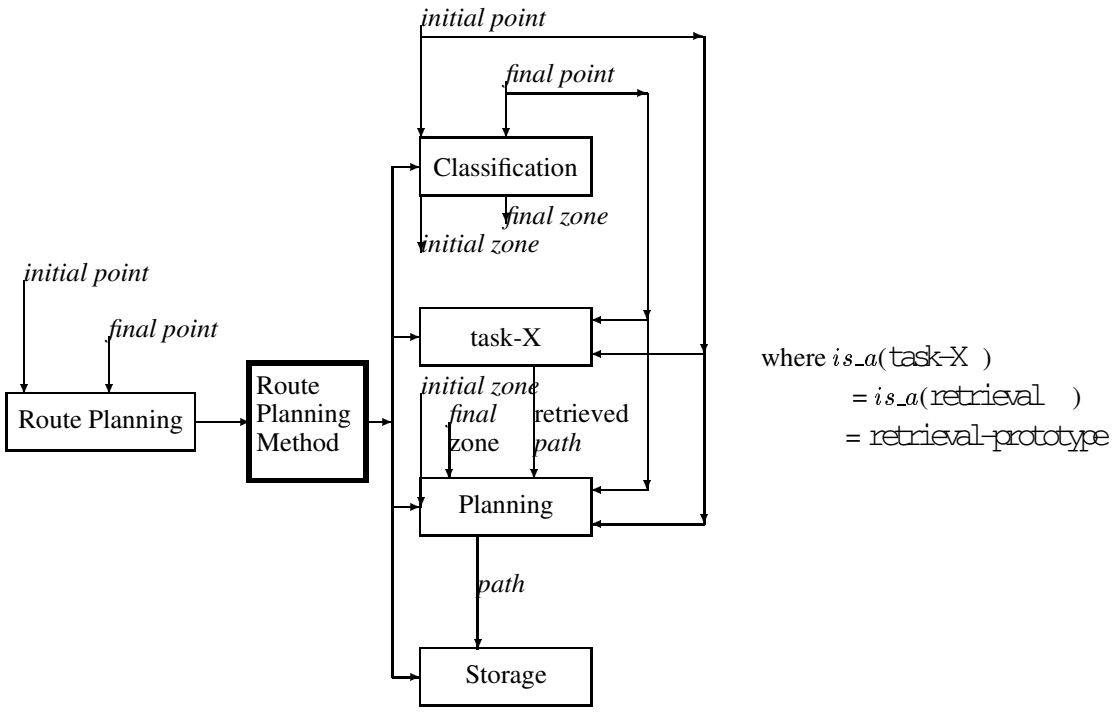


Figure 7.9: ROUTER's task structure after the retrieval task substitution.

AUTOGNOSTIC uses this knowledge, along with the specific values (actual and preferred) of the information type for which the task functionality needs to be modified, to discover such a relation. The process of discovering a relation to unify the current class of values produced for some type of information with the additional values desired for this type (or to distinguish among the desirable and the undesirable values of a type of information) is described in detail in section 7.7. However, it is important to notice that if the system does not have any understanding about the types of knowledge available to it, it cannot modify the information-transformation roles that its existing tasks play.

Assuming that AUTOGNOSTIC has identified such a relation, the next step is to replace the over-constraining task semantics with the newly found relation (or, it is an under-constrained task to add this relation to its semantics). Finally, AUTOGNOSTIC has to modify the procedure which carries out the task in question to comply with the new specification of the task functionality. This last step is a "programming" step and is beyond AUTOGNOSTIC's current capabilities. To be able to "automatically" program an operator which can carry out the information-transformation function desired of a task, is an open research issue in itself. Thus, to date the last step is performed at the suggestion of AUTOGNOSTIC, by a human programmer. The algorithm for extending the semantics of a task is shown in Figure 7.10.

Example Problem 7: Consider the problem-solving episode of example 7 where AUTOGNOSTICONROUTER substituted one instance of the retrieval-prototype with another in the route-planning task structure, in order to extend the class of paths that ROUTER could retrieve and reuse from its memory. If AUTOGNOSTICONROUTER were not aware of any other instantiations of the retrieval-prototype, the task-substitution plan would become inapplicable. An alternative plan applicable to this type of failure cause, is the task-modification plan, in that

MODIFYING-A-TASK-FUNCTIONALITY($flag, t, rel_{violated}, i, v_{actual}, v_{alt}$)

Input:
a flag signifying whether the modification is to extend or to refine the task functionality, $flag$,
the task whose functionality is to be modified, t,
the violated semantic relation (if the modification is an extension), $rel_{violated}$,
the output of the task, i, with respect to which its functionality needs to be modified,
its actual value in the failed problem-solving episode, v_{actual}, and
its desired value v_{alt}.

Output:
a modified task t.

---

(1) If $flag$=$EXTEND$
then discover relation $rel : rel(v_{alt}) \wedge \not rel(v_{actual})$
else discover relation $rel : rel(v_{alt}) \wedge rel(v_{actual})$

(2) If $flag$=$EXTEND$
then add rel in $s(t)$
else replace $rel_{violated}$ with rel in $s(t)$

(3) Replace procedure $op(t)$ with procedure producing all values legal under new $s(t)$

Figure 7.10: Modifying (Specializing/Generalizing) the functionality of a task t.

case, the extension of the retrieval semantics in order to allow the production of the desired value for its current input. The relation that AUTOGNOSTIC discovers as a possible replacement for the failing semantics of the retrieval subtask, is that zone-intersections(initial-zone initial-node(middle-path)) . At this point, it proceeds to replace the current operator of the retrieval task with one that will meet the new desired functionality.

This modification results in the extension of the class of problems that ROUTER is able to solve using its case-based method. Notice, that this extension does not occur because of acquisition of new domain knowledge like in the example discussed in the section 7.4.1. Rather, it occurs because of the modification of an elementary operation of the problem solver, i.e., retrieval . AUTOGNOSTIC has recognized the potential of this operation to perform a more general information transformation than the one it was originally designed for, and, furthermore, it postulated what such a more general transformation should be.

To summarize, in order to modify the retrieval task in ROUTER's task structure, AUTOGNOSTIC performed the following steps:

1. discovered a relation unifying the set of values currently produced by the task for the output middle-path with the new desired one,
2. replaced the failing semantic relation of the task with the new one which covers the desired value for the middle-path , and
3. created a function to carry out the transformation of the new task.

7.5.4 Introducing a Selection Task in the Task Structure

When the functionality of a task is under-constrained, i.e., when this task can produce several values for some of its output information, some of which are not acceptable, a plan which can potentially remedy the problem, is the *plan of a selection task insertion*. The motivation behind the invocation of this plan is to "tailor" the problem-solver's task structure towards producing a narrower, preferred, class of values for this type of information. The blame-assignment process suggests the refinement of a task functionality, $t_{producing}$, when it notices that its semantics allow the production of a range of values for an output information type, i, and some of the values in this range lead to desired solutions while others lead to undesirable ones. A selection task, inserted in the problem-solver's task structure after $t_{producing}$, has as a goal to reason about the possible values of i in the context of a specific problem and select the most appropriate one for the given problem. Thus, the selection-task insertion implies the discovery of a characteristic property of the desired values of i which will distinguish them from the undesirable ones.

As shown in Figure 7.11 which describes the algorithm for inserting a selection task in the problem-solver's task structure, the first step (line 1) of the insertion process is to discover a differentiating criterion between the actual value v_{actual}, and its desired one, v_{alt}, of the information type, i. Assuming that such a relation is discovered, then AUTOGNOSTIC can use it as the semantic relation of the new task to be inserted (line 4). Next, AUTOGNOSTIC creates a new type of information, $i_{intermediate}$, which will hold all the possible values allowed by the semantics of the task which until now produced i, $t_{producing}$ (line 3). This new type of information, $i_{intermediate}$, will now replace i in the output of $t_{producing}$ (line 7), and the procedure accomplishing the failing task must be appropriately modified to produce all its possible values instead of only one (line 8). At this point, AUTOGNOSTIC has a complete specification for the new task, $t_{selecting}$. The final step AUTOGNOSTIC has to perform is to actually integrate the $t_{selecting}$ task in the task structure. To do that, AUTOGNOSTIC creates a new, higher-level task, $t_{overall}$, in the service of which both $t_{producing}$ and $t_{selecting}$ will be performed from now on (line 2). More specifically, AUTOGNOSTIC creates a method, $m_{overall}$, for $t_{overall}$ and sets up as its subtasks, $t_{producing}$ and $t_{selecting}$ (line 5). Finally, the $t_{overall}$ now replaces $t_{producing}$ in its role in the task structure (lines 6, 9).

Example Problem 1: In example 1, AUTOGNOSTICONROUTER was presented with the problem of planning a route from *(10th & center)* to *(ferst-1 & dalney)*. For this problem, AUTOGNOSTICONROUTER produced the path *((center 10th) (10th atlantic) (atlantic ferst-1) (ferst-1 dalney))* and was presented as feedback the path *((center 10th) (10th dalney) (ferst-1 dalney))*. In this scenario, the blame-assignment process suggested as a possible cause of its failure the under-constrained functionality of ROUTERs classification subtask.

Given the type of the cause of the failure, and the knowledge conditions at the time, (i.e., there are no tasks that can potentially substitute the current classification task, and it produces several types of output information), the repair subtask selects as an applicable plan the insertion of a selection task after classification which will reason about the possible values of the initial-zone and select the appropriate one. In the context of the current failed problem-solving episode, the appropriate value would be *za* as opposed to *z1* which was actually produced. The actual and the alternative values for initial zone in this example, *z1* and *za*, will constitute the "pool" of possible values output by the classification task which in the new task structure will be a new type of information, called intermediate-initial-zone. This new type will constitute (part of) the input for the new selection-after-classification task which from now on will produce the initial zone.

The information initial zone is an instance of the domain concept zone. AUTOGNOSTIC knows that one domain relation applicable to this type of objects is the children-zones relation, where children-zones(n1 n2) means that the *n2* is a sub-neighborhood of *n1*. AUTOGNOSTIC discovers that ForAll n in intermediate-initial-zone children-zones(n initial-zone). Thus it hypothesizes that this can be used as a differentiating criterion between possible alternative values for the initial-zone. At this point AUTOGNOSTIC has a complete specification for the new selection task selection-after-classification: its input will be intermediate-initial-zone, its output will be initial-zone and its

---

INSERT-SELECTION-TASK($i, t_{producing}, v_{actual}, v_{alt}$)

Input:
the task after which the selection task will be inserted, $t_{producing}$,
the output of the task, i, which needs to be selected,
its actual value in the failed problem-solving episode, v_{actual}, and
its desired value v_{alt}.

Output:
a modified task structure, with a new task in it.

---

(1) Discover relation $rel : rel(v_{alt}) \wedge not(rel(v_{actual}))$

(2) Create task $t_{overall}$: $i(t_{overall}) = i(t_{producing}) \cup \{info1(rel)\ info2(rel)\} - \{i\}$

$o(t_{overall}) = \{i\}$

$by(t_{overall}) = m_{overall}$

(3) Create information type $i_{intermediate}$: $wo(i_{intermediate}) = wo(i)$

$syntactictype(i_{intermediate}) = multiple$

(4) Create task $t_{selecting}$: $i(t_{selecting}) = \{i_{intermediate}\} \cup \{info1(rel)\ info2(rel)\} - \{i\}$

$o(t_{selecting}) = \{i\}$

$s(t_{selecting}) = \{rel\}$

(5) Create method $m_{overall}$: $subtasks(m_{overall}) = \{t_{producing} t_{selecting}\}$

$control(m_{overall}) = \{serial(t_{producing} t_{selecting})\}$

(6) Replace t in $subtasks(subtaskofmethod(t))$ with $t_{overall}$

(7) Replace i in $o(t)$ with $i_{intermediate}$

(8) Replace procedure $op(t)$ with a procedure producing all values legal under the semantics $s(t)$

(9) Replace t in $producedby(i)$ with $t_{overall}$

Figure 7.11: Inserting a selection task to deliberately reason about the possible values of i.

semantics will be ForAll n in intermediate-initial-zone children-zones(n initial-zone) .

AUTOGNOSTIC, at this point, creates an overall-classification task and a new method, overall-classification-method which decomposes the above task into a pair of sequentially ordered subtasks: classification and selection-after-classification . This new task, overall-classification , now replaces the classification task in the set of subtasks of route-planning-method .

In our example, selecting the most specific value for the initial-zone results in the selection of a low-level neighborhood. Given that lower-level neighborhoods describe smaller spaces in more detail, ROUTER's search becomes very local, and the two problem locations are connected through small pathways instead of major ones, which, in general, results in shorter paths.

To summarize, in order to introduce the new task in ROUTER's task structure AUTOGNOSTIC performed the following steps:

1. it introduced a new type of information intermediate-initial-zone to hold the intermediate results of the classification task and to be the input of the new task,

2. it created a procedure to carry out the transformation of the new task,
3. it changed (had it reprogrammed by an oracle) the procedure classification-func to actually return appropriately a list of values instead of a single one,
4. it created a new higher-level task, overall-classification, and a new method for it, overall-classification-method, in the service of which the classification and the selection-after-classification task will be performed, and
5. it modified the route-planning method to include overall-classification in the place of the classification task.

The modified part of ROUTER's task structure is shown in Figure 7.12.

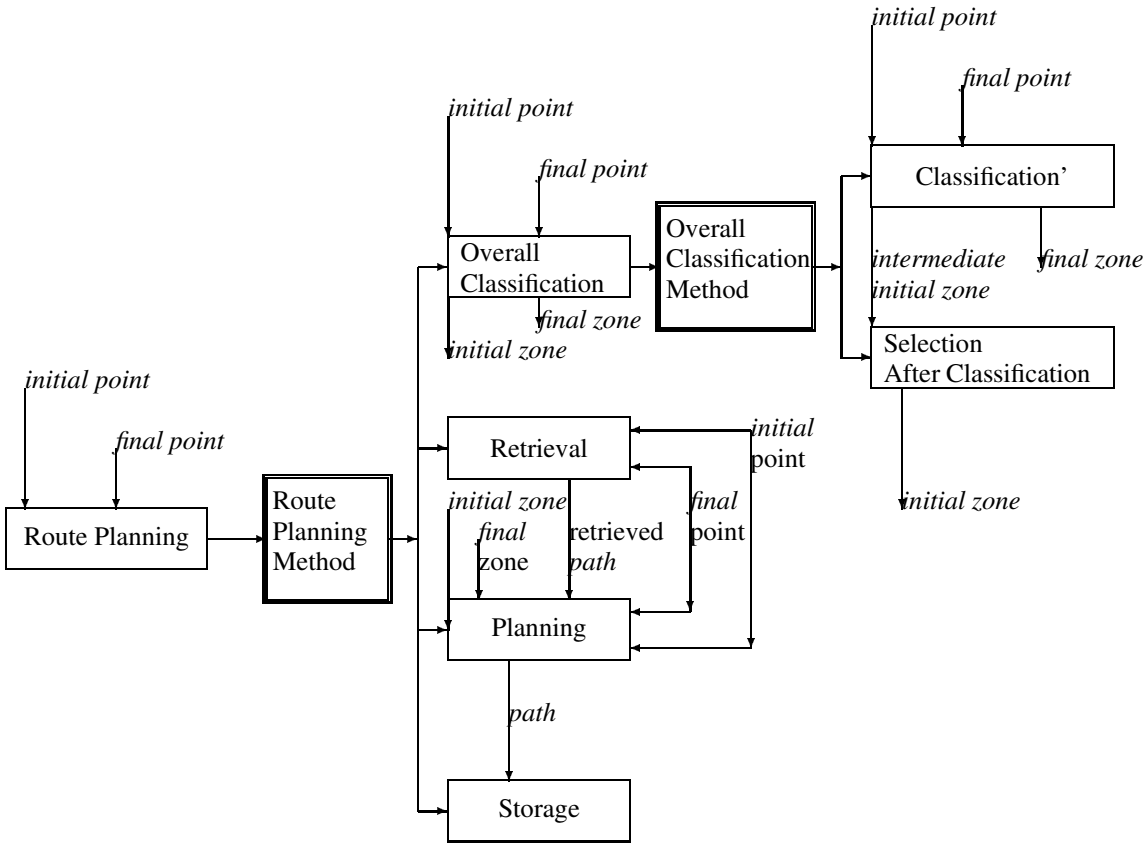


Figure 7.12: Inserting a selection task in ROUTER's task structure.

7.6 Modifications to the SBF Model of the Problem Solver

AUTOGNOSTIC's reflective learning process is based on its comprehension of the problem-solving process as it is specified in AUTOGNOSTIC's SBF model of the problem solver. Learning based on the problem-solver's SBF model presents many advantages to reasoning based simply on the

problem-solver's trace. The major ones among these advantages are the system's ability first, to address the more general blame-assignment task which admits the possibilities that both the domain-knowledge and the task structure might be incorrect, and second, to modify its own process in a way that maintains its consistency and integrity. However, it comes with an additional cost: namely the cost of maintaining the integrity of the SBF model of the problem solver with respect to the actual problem solver itself.

To date, AUTOGNOSTIC assumes the existence of the problem-solver's SBF model. It does not assume however that this model is correct. AUTOGNOSTIC is able to recognize inconsistencies between the SBF model and the actual behavior of the problem solver while reasoning on a particular problem. When each subtask of the problem solver is completed, AUTOGNOSTIC evaluates whether its actual product verifies the semantics which characterize the information transformation of the task. If it does not, and if the overall product of the problem-solving process is acceptable, AUTOGNOSTIC infers that the SBF model misrepresents the problem-solver's behavior and sets up as its goal the modification of the SBF model, so that it matches the actual process of the problem solver. This modification is carried out by the *plan for semantics revision*.

Figure 7.13 depicts the algorithm for this type of modification. Notice, that it is similar to the algorithm for modifying the functionality of an over-constrained task: they both are triggered by a violated semantic relation, and they both aim in the substitution of this relation with another. The difference between the two, is that in the case of the task-semantics extension, the values violating the semantic relation are the alternative values that can potentially lead to the production of the feedback, where in the case of the task-semantics redefinition, the values violating the semantic relation are the values actually produced in the course of problem solving. Also, in the first case, AUTOGNOSTIC has to modify the procedure carrying out the task under inspection so that it meets the new semantic relation. In the second case, AUTOGNOSTIC does not need to modify the procedure since the purpose of this modification is to capture the nature of the actual transformation. The plan of task-semantics redefinition can be invoked with non-leaf tasks. Finally, it is important to notice that this modification only affects the quality of AUTOGNOSTIC's introspective integrity, that is the quality of its SBF model with respect to the object-level problem solver. It does not have any affect to the problem-solving mechanism itself.

REDEFINING-TASK-SEMANTICS($t, rel_{violated}, i, v_{actual}$)

Input:

the task whose semantics need to be modified, t,
the violated semantic relation, $rel_{violated}$,
the output of the task to which the violated semantic relation applies, i
its actual value in the failed problem-solving episode, v_{actual}, and
its desired value v_{alt}.

Output:

a modified task structure, with a new task in it.

---

(1) Discover relation $rel : rel(v_{alt}) \wedge rel(v_{actual})$

(2) Replace $rel_{violated}$ in task semantics $s(t)$ with rel

Figure 7.13: Redefining the semantics of task t so that it correctly specifies the behavior of task t.

Example Problem 2: While solving the example problem 2[2] of going from *(ferst-1 & hemphill)* to *(home-park & atlantic)*, AUTOGNOSTICONROUTER notices that the values produced by its retrieval subtask for the middle-path do not conform with its semantics: the initial-node of the retrieved path, *((10th center) (10th atlantic) (home-park atlantic))*, is not the same as the initial-point of the current problem. The blame-assignment process suggests as a potential cause for this failure that the semantics of the retrieval subtask are wrong and are violated by its behavior, although this behavior is actually correct. The repair process then identifies as the appropriate adaptation the redefinition of the failing semantic relation.

While actually effectuating the suggested modification, AUTOGNOSTICONROUTER discovers that all its problem-solving episodes meet the relation zone-intersections(initial-node(middle-path) initial-zone), and therefore it infers that it can substitute the failing semantic relation, (same-point initial-node(middle-path) initial-point), with this one. Notice, this is exactly the same relation it discovered in the process of effectuating the extend-task-semantics modification in the example problem 7. However, the affects of these two modifications to the system's behavior are different.

7.7 Discovering New Semantic Relations

AUTOGNOSTIC's ability to modify its task structure (with the exception of the task-reorganization modification) relies on its ability to discover new relations, which can be used either to further characterize the information-transformation functions of its existing tasks, or to postulate as functions for new tasks that should be introduced in its its task structure. For example, both the plan for modifying an existing task and the plan for inserting a task in the task structure include a "discover semantic a relation" step. In the former plan, the goal is to infer a relation which will characterize the new functionality of the modified task. In the latter plan, the goal is to postulate a functionality for a new task altogether. This ability of AUTOGNOSTIC's is based on its comprehension of the domain knowledge generally available to the problem solver.

The SBF modeling framework is based on an information processing ontology. In the SBF language, tasks are inferences, i.e., information-transformation functions, and methods are rules for composing elementary tasks into more complex ones, or equivalently, elementary inferences into more complex ones. Finally, the knowledge provides the "medium" to which the information-transformation functions are applied (i.e., inferences are about the different types of concepts relevant in a domain), and also, it defines the range of inferences that are possible in a particular domain (inferences relate instances of domain concepts, and thus the possible types of inferences are bounded by the types of domain relations applicable in a domain). By having an explicit understanding, on one hand, of the inferences that are potentially involved in the problem-solver's reasoning (the tasks in its task structure), and on the other, of the range of potential inferences that its domain knowledge can be used for, the system is able to recognize new uses for its knowledge, and thus to integrate new inferences in its task structure.

The process for discovering a new relation to be used either as a new semantic relation for an existing task, or as the semantic relation of a new task, is invoked by the plans for modifying the functionality of a task and for introducing a selection task. It is also invoked by the plan for redefining the semantics of a task when they fail to express the problem-solver's actual behavior in a particular problem.

In situations where the task-modification plan is applied to refine the task functionality, or where the insertion of a selection task plan is invoked in order to tailor some output of the task, the goal of the relation-discovery process is to produce a relation which will differentiate the value actually produced during problem solving for some type of information, from the value desired for this information, i.e., the value which will enable the production of the feedback.

When the task-modification plan is applied to extend the functionality of the task, the goal of the process is to produce a relation unifying the value desired for a type of information with the

[2]This discussion is based on the knowledge context of ROUTER as discussed in section 5.2.2 of chapter 5.

value that was actually produced during problem solving for this information.

Finally, in the case of redefining the semantics of a task when they fail to describe the actual behavior of the task in the context of a particular problem-solving episode, the goal is also to produce relation characterizing the values that the task at hand produces for some type of information, which does not get violated by the problem-solver's behavior in any actual episode.

Figures 7.14 and 7.15 depict the algorithms for discovering a new relation which characterizes a set of examples, or which differentiates a set of examples from another set of examples. These algorithms are symmetrical, and they consist of five basic steps:

1. AUTOGNOSTIC identifies the types of information, available to the problem solver, in its immediate information context, before the accomplishment of the task at hand (line 0).

2. AUTOGNOSTIC identifies a set of domain relations that relate the type of information in question with other available types of information (line 1).

3. It identifies a set of relations in the task structure (semantics or conditions) that are applicable to types of information which are instances of the same type of domain object as the information in question (line 2).

4. It instantiates both these types of relations (lines 3a-3b, 3d-3e) in the current context; that is, it creates new semantic relations that relate the type of information in question with other types of information available in the context where the modification was raised.

5. It evaluates (lines 3c, 3f) which of these instantiations actually play the role they are hypothesized for, i.e., differentiation or unification of a set of examples.

6. Finally, if no relation has been found at that level, AUTOGNOSTIC attempts to identify relations characteristic of the attributes of the information in question (line 4).

The first step sets up the hypothesis space in which AUTOGNOSTIC will search for the desired relation. Essentially, it identifies the set of possible attributes in terms of which the new functional concept can be described. Attribute or term selection is a very important problem in concept learning, and traditionally, it has been the responsibility of the system designer to decide which the potentially useful attributes are. AUTOGNOSTIC is able to provide an initial answer for the specific instance of this problem it is dealing with: namely, when the problem is to learn functional concepts, the "attributes" in terms of which the desired concept will be expressed, belong in the set of types of information available to the problem solver at this point in its task structure where this new functional concept will be introduced. Furthermore, because AUTOGNOSTIC can potentially maintain a set of nested information contexts, it selects the most immediate one in order to bound the hypothesis space around the most relevant types of information. AUTOGNOSTIC's ability to select the attributes for each new concept lies in its comprehension of the information interactions among the different subtasks in the problem-solving task structure, as captured in the SBF model of the problem solver. It is important to notice that, this set of attributes is context-dependent, i.e., depending of where in the problem-solving process the new concept is needed a different set of attributes are available for describing it. This sensitivity to the problem-solving context where the new concept will be integrated bounds the dimensions of the concept space and consequently limits the search for the right concept.

Example Problems 2 and 7: In the modifications that AUTOGNOSTIC performed in response to the failures that occurred in the process of solving example problems 2 and 7, the same unifying relation was discovered. Let us discuss in detail the process by which AUTOGNOSTIC concludes that this is an appropriate relation, in both these cases.

In both cases, AUTOGNOSTIC is searching for a relation which pertains to paths, since it is trying to produce a relation to qualify the middle-path produced by the retrieval task. Thus, it first identifies which of the domain relations it knows about refer to paths. AUTOGNOSTIC does not find any such relations.

UNIFYING-RELATION-DISCOVERY($i, cases, t_{overall}$)

Input:
the type of information which the relation must characterize, i,
a set of examples, $cases$,
i.e., problem-solving episodes, in which the values of i are desirable, and
the overall task, $t_{overall}$,
in the service of which, the task characterized by the discovered relation will be invoked.

Output:
a relation characterizing the values of i in all the given $cases$.

---

(0) $available\_info$:= $\{i : \exists t_p : i \in o(t_p) \wedge t_p$ is accomplished before t

(1) $domain\_rels$:= $relations(wo(i))$

(2) old_ts_rels := $\{rel : rel \in s(t)\ t a task \in subtree(t_{overall})$
$\vee rel \in c(m)\ m$ a method $\in subtree(t_{overall})\}$

(3a) $\forall rel \in \{domain\_rels \cup old\_ts\_rels\}$
If $type(rel)$ = $domain\_rel$
then $wo2$:= $input\_args(rel) \cup output\_args(rel) - wo(i)$

(3b) $\forall i2 \in cases : wo(i2)$ = $wo2$,
instantiate rel into $rel' : type(rel')$ = ts_rel with i and $i2$

(3c) **If** $\forall c \in cases\ \ true(rel'(v(i)_c\, v(i2)_c))$
then include rel' in $appropriate\_rels$

(3d) **If** $type(rel)$ = ts_rel
then $wo2$:= $\{wo(info1(rel))\ wo(info2(rel))\} - wo(i)$

(3e) $\forall i2 \in cases : wo(i2)$ = $wo2$,
instantiate rel into $rel' : type(rel')$ = ts_rel with i and $i2$

(3f) **If** $\forall c \in cases\ \ true(rel'(v(i)_c\, v(i2)_c))$
then include rel' in $appropriate\_rels$

(4) **If** $appropriate\_rels$=$\emptyset$
then $\forall attr \in attr(i)$
$appropriate\_rels$:= unifying-relation-discovery($attr, cases, t_{overall}$)

Figure 7.14: Discovering a unifying relation rel.

DIFFERENTIATING-RELATION-DISCOVERY ($i, cases_{positive}, cases_{negative}, t_{overall}$)

Input:
the type of information which the relation must characterize, i,
a set of positive examples, $cases_{positive}$,
i.e., problem-solving episodes, in which the values of i are desirable,
a set of negative examples, $cases_{negative}$,
i.e., problem-solving episodes, in which the values of i are incorrect, and
the overall task, $t_{overall}$,
in the service of which, the task characterized by the discovered relation will be invoked.
Output:
a relation characterizing the values of i in all the given $cases$.

---

(0) $available\_info$:=$\{i : \exists t_p : i \in o(t_p) \wedge t_p$ is accomplished before t

(1) $domain\_rels$:=$relations(wo(i))$

(2) old_ts_rels :=$\{rel : rel \in s(t) : tatask \in subtree(t_{overall})$
$\vee rel \in c(m) : m$ a method $\in subtree(t_{overall})\}$

(3) $\forall rel \in \{domain\_rels \cup old\_ts\_rels\}$

(3a) If $type(rel)$ =$domain\_rel$
then $wo2$:=$input\_args(rel) \cup output\_args(rel) - wo(i)$

(3b) **(1)** $\forall i2 \in cases : wo(i2)$ =$wo2$,
instantiate rel into rel' : $type(rel')$ =ts_rel with i and $i2$

(3c) **If** $\forall c \in cases_{positive}\ \ true(rel'(v(i)_c\, v(i2)_c))$
$\wedge \forall c \in cases_{negative}\ \ false(rel'(v(i)_c\, v(i2)_c))$
then include rel' in $appropriate\_rels$

(3d) If $type(rel)$ =ts_rel
then $wo2$=$\{wo(info1(rel))\ wo(info2(rel))\} - wo(i)$

(3e) $\forall i2 \in cases : wo(i2)$ =$wo2$,
instantiate rel into rel' : $type(rel')$ =ts_rel with i and $i2$

(3f) **If** $\forall c \in cases_{positive}\ \ true(rel'(v(i)_c\, v(i2)_c))$
$\wedge \forall c \in cases_{negative}\ \ false(rel'(v(i)_c\, v(i2)_c))$
then include rel' in $appropriate\_rels$

(4) If $appropriate\_rels$:=$\emptyset$
then $\forall attr \in attr(i)$
$appropriate\_rels$:= diff-relation-discovery($attr, cases_{positive}, cases_{negative}, t_{overall}$)

Figure 7.15: Discovering a differentiating relation rel.

Next it searches for semantic relations of other tasks in its task structure, which produce as input and/or consume as output information a kind of path. Because the semantic relations of the problem-solver's tasks need not refer only to domain relations, but they can also refer to more complicated predicates, AUTOGNOSTIC collects the existing semantic relations referring to paths, in order to investigate whether they may be used to play yet another role in the task structure, i.e., as semantic relation of the *retrieval* task. In this step, AUTOGNOSTIC indeed collects four potential relations, originally annotating the *path-synthesis* task[3]: *first-subpath(path? desired-path)*, *second-subpath(path? desired-path)*, *third-subpath(path? desired-path)*, and *length(desired-path) > 6*.

Next AUTOGNOSTIC proceeds to instantiate these relations in its current context. The first three relations cannot be instantiated in the context of the *retrieval* task, because they all require another information which must be of the type *path*, and there is no other path already produced at the time when the *retrieval* task is performed. The fourth one can be instantiated, (since it is a unary relation), but it does not actually reflect the types of paths that *retrieval* produces. That is, it is not true for the output paths of *retrieval* and thus it is rejected.

At this stage, AUTOGNOSTIC starts to investigate the possibility that the relation which can appropriately characterize the paths produced by the *retrieval* task[4] may refer to some attribute of that path. Thus it starts the process of a unifying-relation discovery again for the attributes *initial-node* and *final-node* of the retrieved path. Since these two attributes are intersections, and there are several domain relations in ROUTER's domain that refer to intersections, AUTOGNOSTIC is able to collect at its first step a range of candidate relations, including *zone-intersections*, *zones-of-int*, and *same-point*. These relations relate intersections to zones, zones and other intersections respectively. There are two different types of zones, and two different types of intersections in the context of *retrieval* task: *initial-zone* and *final-zone*, and *initial-point* and *final-point* respectively. AUTOGNOSTIC instantiates these domain relations with the available types of information, and after evaluating the instantiated relations, it concludes that two of them can be potentially used: *zone-intersections(initial-zone initial-node(middle-path))*, or *inverse(zones-of-int) (initial-zone initial-node(middle-path))*.[5] Of the two, AUTOGNOSTIC prefers the former, since it is simpler.

7.7.1 Discussion

AUTOGNOSTIC's task of discovering a new relation with which to characterize the information-transformation intended by some task is a kind of learning a concept from examples. For a particular failed problem-solving episode, AUTOGNOSTIC's blame-assignment process, based on the SBF model of the problem solver under consideration, identifies the inferences that this problem solver should have made, in order to produce the desired solution. These inferences are instances of the abstract classes of information transformations that the problem-solver's subtasks should be able to perform in order to produce the desired class of solutions for the its overall task. In some cases, namely the cases where AUTOGNOSTIC recognizes as a potential cause of the failure the under-constrained functionality of some task, these inferences are congruent with the currently expected behaviors of the corresponding subtasks as described by the semantics of these subtasks. In these cases, the intended behaviors of these subtasks, i.e., its semantics, need to be more sharply characterized so that the preferred inferences are positive examples of the class of information-transformations these tasks perform, while the actual inferences, made during the failed

[3] The *path-synthesis* task is the prototype of both *mbr-path-synthesis* and *dbr-path-synthesis*.

[4] Actually, in example problem 2 the goal is to identify the types of paths that *retrieval* is producing, where in the context of the example problem 7, the goal is to characterize the paths that this task should be able to produce in order for the overall problem-solving process to be able to produce the desired path presented as feedback to AUTOGNOSTICONROUTER.

[5] The relations *zone-intersections* and *zones-of-int* are inverse of each other; the first maps an intersection to the neighborhoods it belongs, and the second indexes of a neighborhood the intersection that belong in it. This is why the *inverse* in the second case.

problem-solving episode, are not. In these cases, the task of discovering a new semantic relation is, in fact, equivalent to concept specialization based on one negative and one positive example.

In other cases, namely the cases where AUTOGNOSTIC recognizes as a potential cause of the failure the over-constrained functionality of some task, the desired inferences are in conflict with the current semantics of the subtasks that should have made them. In these cases, the intended behaviors of these subtasks, need to be extended so that both the actual and the preferred inferences are positive examples of the class of information-transformations these tasks perform. In these cases, the task of discovering a new semantic relation is equivalent to concept generalization based on two positive examples.

There are several interesting aspects to the concept learning task as it occurs in the context of AUTOGNOSTIC's reflective learning:

1. the learner's goal is to characterize a task, that is, to specify at some abstract level a class of inferences desired of the problem solver,

2. there is no formulation of the target concept known a-priori, and

3. the language in which the examples are expressed and the language of the target concepts are different.

AI research on concept learning [Mitchell *et al.* 1981, Quinlan 1986, Martin 1992], with the exception of Bacon [Langley 1980] which learns relations between numeric variables and explanation-based methods used to learn problem-solving control heuristics [DeJong and Mooney 1986, Mitchell *et al.* 1986], has focused mainly on learning concepts of objects. AUTOGNOSTIC's learning task has as a goal to characterize a class of inferences that are desired of a problem solver. Once this class of desired inferences has been characterized, AUTOGNOSTIC modifies the problem-solver's task structure so that one of its existing subtasks (or a newly introduced subtask) can draw the inferences of this desired class for each specific problem the problem-solver is presented with.

Another important aspect, with respect to which it is interesting to characterize learning methods, is the a-priori knowledge these methods have regarding the concepts they need to learn. Inductive methods [Mitchell *et al.* 1981, Quinlan 1986] assume only a syntactic description of the target concept. On the other hand, explanation-based methods [DeJong and Mooney 1986, Mitchell *et al.* 1986] assume that a non-operational description of the target concept is already known. AUTOGNOSTIC does not have any a-priori hypotheses regarding the concepts it might need to learn. The space of the concepts that AUTOGNOSTIC may learn is defined syntactically, by the expressiveness of the language in which the learner describes the semantics of its tasks. To date, AUTOGNOSTIC uses a language which allows unary and binary relations, with unique predication, and with unique qualification. The grammar of AUTOGNOSTIC's language for describing semantics is shown in Figure 7.16.

Having only a syntactic specification of the target concept, an important issue that arises is to define the vocabulary in terms of which the target concept may be expressed. This is also known as the problem of attribute selection. The examples based on which AUTOGNOSTIC learns its functional concepts are examples of actual and/or desired problem-solving contexts. Each time AUTOGNOSTIC identifies an over-(under-)constrained task in the problem-solver's task structure, it collects all the information available to the problem solver at this point in its reasoning process where the failed task is performed. All these types of information may be instances of different types of domain objects, and each one may have several different attributes. These information types and their attributes provide the terms in which the target concept will be described. For all these different types of information AUTOGNOSTIC knows their actual values during the failed problem-solving episode, and for some of them it may also know their desired values, i.e., these values that would have led to the production of the desired solution. These actual and desired values provide the examples that the concept must cover. At that point, AUTOGNOSTIC simply generates hypotheses and tests them against the examples, and collects all the plausible hypotheses that can be expressed in its language of task semantics.

After having collected a set of concepts that can be used to characterize the desired information transformation, AUTOGNOSTIC selects one of them and integrates it in the problem-solver's task

```
Relation      := U-Relation || B-Relation
U-Relation    := Q-U-Relation || S-U-Relation
Q-U-Relation  := (Q, F, attr1, info1)
S-U-Relation  := (F, attr1, info1)
B-Relation    := Q-B-Relation || S-B-Relation
Q-B-Relation  := (Q, F, attr1, info1, attr2, info2)
S-B-Relation  := (F, attr1, info1, attr2, info2)
Q             := { (for-all info1), (for-all info2), (there-is info1), (there-is info2) }
```

Figure 7.16: The grammar of AUTOGNOSTIC's language for task semantics.

structure. The criterion for selecting among the potentially several candidates is similar to the "systematicity principle" [Gentner 1989]. More specifically, AUTOGNOSTIC prefers these candidates which provide the maximum number of connections among the different types of information in the problem-solving context. Thus, it prefers binary relations over unary ones, and universally qualified relations over unqualified ones. The chosen relation is integrated in the task structure as the semantic relation of an existing task substituting a failed semantic relation, or as the semantic relation of a new task. The other candidates are instantiated as its "sister" tasks, i.e., tasks instantiations of the same prototypical task. If the chosen concept is not the correct one, AUTOGNOSTIC may substitute it at a later time with one of the other candidates.

7.8 Summary

To summarize, AUTOGNOSTIC is able to

1. acquire new domain knowledge (by acquiring knowledge of new instances of domain objects, and by updating the contents of its state-of-the-world domain relations),
2. reorganize its domain knowledge (by updating the contents of its convention domain relations),
3. reorganize the problem-solver's task structure,
4. introduce new subtasks in the overall task structure of the problem-solving mechanism (by substituting existing failing tasks with new ones, and by inserting selection tasks to tailor its reasoning process towards the desired kinds of solutions), and
5. modify the elementary tasks of the problem-solving process (by extending or refining the information transformation they perform).

AUTOGNOSTIC's reflective learning process is able to improve and expand the domain knowledge of the problem solver (i.e., modifications 1 and 2), to improve the flow of information and control in the problem-solver's task structure by reorganizing its subtasks and by better utilizing its methods (i.e., modification 3), and finally, to repair the elementary functional elements of the problem-solving process so that it addresses the desired class of problems and produces solutions of the desired quality (i.e., modification 4 and 5).

CHAPTER VIII

A REFLECTION EPISODE WITH AUTOGNOSTIC

Chapter 1 raised a set of issues involved in the problem of failure-driven learning. Chapter 2 discussed the major subtasks of a failure-driven learning process and analyzed the types of knowledge they need to draw the inferences intended of them in the context of the overall process. Chapter 3 described AUTOGNOSTIC's SBF language and discussed the knowledge that the model of a problem solver, expressed in this language, captures. Chapters 5, 6, and 7 explained the processes by which each subtask of AUTOGNOSTIC's reflective failure-driven learning process in detail. These chapters illustrated the individual processes with excerpts from AUTOGNOSTIC's reflective behavior on several example problems. This chapter discusses a complete reflective problem-solving-and-learning episode, presents a complete trace of AUTOGNOSTICONROUTER's reasoning in that episode, revisits the issues that each step of the process has to address, and discusses AUTOGNOSTIC's answers to these issues.

This trace is from AUTOGNOSTICONROUTER as it solves example problem 3 of going from *(10th center)* to *(ferst-1 dalney)*. The discussion in this chapter makes references to the SBF model of ROUTER, which can be found in Appendix 1. To make the example easier to follow, let us describe the conventions used throughout the presentation of AUTOGNOSTICONROUTER's trace. The major subtasks of the reflective process and their specific inputs for this episode are shown boxed at the point in the process where this task is spawned. The outputs of these subtasks are also shown boxed at the point in the process where the task has been accomplished. During problem solving and monitoring, AUTOGNOSTIC's inferences (inferences in the meta-reasoning space) are introduced by "Autognostic", where ROUTER's inferences (inferences in the reasoning space) are introduced by "Router". The trace of AUTOGNOSTICONROUTER's reasoning is split in segments, each of which is followed by a discussion on the progress of the system's reasoning in the trace segment above it.

8.1 Monitoring

Monitoring the problem solving constitutes the first step of the reflective process. While solving the problem AUTOGNOSTICONROUTER also monitors its progress. The role of the monitoring process is

1. to evaluate the progress of the problem-solving process,
2. to actively look for failures in this process, and
3. to keep a record of all the problem-solver's elements involved in the process and their contributions, which will define the space of possible hypotheses for the cause of the failure, in case there is one.

The input to the monitoring step, as shown in the algorithm of Figure 5.1, is the task to be accomplished, and the information context in which it is to be accomplished. In the case of this episode the task to be accomplished is route-planning and its information context is { (initial-point *(10th center)*)(final-point *(ferst-1 dalney)*)(top-neighborhood *z1*) }.

```
(monitor task := route-planning
        information context := ( (initial-point (10th center))
                                 (final-point (ferst-1 dalney))
                                 (top-neighborhood z1)))
```

```
Autognostic:    Task to accomplish ROUTE-PLANNING
Autognostic:    Accomplishing Task ROUTE-PLANNING
Autognostic:    Potentially applicable methods: (ROUTE-PLANNING-METHOD)
Autognostic:    The task ROUTE-PLANNING can be accomplished
     by method ROUTE-PLANNING-METHOD
Autognostic:    This method decomposes the task into the subtasks
     (CLASSIFICATION PATH-RETRIEVAL SEARCH STORE)
```

Initially, AUTOGNOSTICONROUTER sets *route-planning* as the task it needs to accomplish, since this is the task specified in its input problem. Next, it proceeds to evaluate whether or not there is sufficient information in the current (which also happens to be the input) context to accomplish this task. From the specification of this task in the SBF model of ROUTER, AUTOGNOSTICONROUTER knows that this task takes as input an initial and a final intersection and the overall neighborhood in which the problem should be solved. AUTOGNOSTICONROUTER examines the current information context, and as all these types of information are specified, it proceeds to accomplish the task. The SBF model specifies that there are two methods for accomplishing this task, one of which, *trivial-route-planning-method*, is applicable under the condition that the initial and final intersections are the same. Thus, AUTOGNOSTICONROUTER concludes that there is only one method potentially applicable in the current episode, *route-planning-method*, which it then proceeds to invoke. The invocation of this method results in the decomposition of *route-planning* into the subtasks *classification*, *path-retrieval*, *search*, and *store*. The *route-planning-method* specifies that these subtasks should be accomplished in a serial order, thus, at this point the focus of the monitoring process shifts to the first of these subtasks, *classification*.

```
Autognostic:    Task to accomplish CLASSIFICATION
Autognostic:    Accomplishing Task CLASSIFICATION
     by invoking the procedure CLASSIFICATION-PROC

     Router:    classifying the intersections in the neighborhood hierarchy
     Router:    output (z1 z1)

Autognostic:    Evaluating the behavior of CLASSIFICATION
     (ZONE-INTERSECTIONS INITIAL-ZONE:=z1 INITIAL-POINT:=(10th center))
     (ZONE-INTERSECTIONS FINAL-ZONE:=z1 FINAL-POINT:=(ferst-1 dalney))
Autognostic:    Task CLASSIFICATION successfully completed
```

Classification is a leaf task in ROUTER's task structure. This means that it is accomplished by a piece of code, non-inspectable by AUTOGNOSTICONROUTER, namely the procedure *classification-proc*. AUTOGNOSTICONROUTER invokes this procedure which returns the data *(z1 z1)*. At this point AUTOGNOSTICONROUTER interprets these data as the output information of the *classification* task, and proceeds to evaluate the data against the functional semantics of this task. Indeed the data meets AUTOGNOSTICONROUTER's expectations regarding the information *initial-zone* and *final-zone*, and thus the information context is updated to include with this new information.

The pointer of each leaf task to the procedure that accomplishes it in the SBF model constitutes a link between the primitive elements in the system's structure and their functional specifications. This link enables AUTOGNOSTICONROUTER to invoke the procedure corresponding to the leaf task at the top of its agenda, and to receive its results and interpret them as the output information expected of the task.

In the above segment, it is evident how AUTOGNOSTICONROUTER evaluates the correctness of the actual behavior of the problem solver in the context of a particular episode. As it accomplishes a leaf task, it applies its functional semantics (if any) to the values this task has produced for its output information. If the specific values satisfy these semantics, then AUTOGNOSTICONROUTER establishes that the specific inference that this task has contributed to the problem-solving process is within the class of inferences it was designed to produce. For a more detailed discussion on the evaluation of a specific task inference, see Section 5.1.

```
Autognostic:    Task to accomplish PATH-REIRIEVAL
Autognostic:    Accomplishing Task PATH-REIRIEVAL
Autognostic:    Potentially applicable methods: (PATH-REIRIEVAL-METHOD)
Autognostic:    The task PATH-REIRIEVAL can be accomplished
     by method PATH-REIRIEVAL-METHOD
Autognostic:    This method decomposes the task into the subtasks
     (REIRIEVAL FIND-DECOMPOSITION-INTS)
Autognostic:    Task to accomplish REIRIEVAL
Autognostic:    Accomplishing Task REIRIEVAL
     by invoking the procedure REIRIEVE-FROM-DOMAIN-BASED-ON-ZONES
     Router: retrieve-path, with input, ((10th center)(ferst-1 dalney) z1 z1)
     Router: output nil

Autognostic:    Task to accomplish FIND-DECOMPOSITION-INTS
Autognostic:    This task does not contribute any information in the current context
```

The next task in AUTOGNOSTICONROUTER's agenda, as set up by the route-planning-method, is path-retrieval. This task can be accomplished (all the input information it requires is available) and AUTOGNOSTICONROUTER proceeds to accomplish it. In the decomposition of path-retrieval, the subtask which actually probes ROUTER's path memory is the retrieval task, which in this episode returns no path at all.

Notice that, in the path-retrieval task decomposition, there is yet another task, find-decomposition-ints. The role of this task is to set up the intersections, int1 and int2 respectively, given a path retrieved from memory. These two intersections, may later be used to specify the input of the subtasks of the case-based method, front-path planning and end-path planning, in case this method is invoked. Due to the nature of this task, the SBF model specifies that it contributes to the progress of the problem solving only under the condition that a path has been retrieved from memory. Because this condition is not met in the current context, AUTOGNOSTICONROUTER does not even attempt to accomplish this task.

```
Autognostic:    Task to accomplish SEARCH
Autognostic:    Accomplishing Task SEARCH
Autognostic:    Potentially applicable methods: (MODEL-BASED-METHOD CASE-BASED-METHOD)
Autognostic:    Applicable methods: (MODEL-BASED-METHOD)
Autognostic:    The task SEARCH can be accomplished
     by method MODEL-BASED-METHOD
Autognostic:    This method decomposes the task into the subtasks
     (INTRAZONAL-SEARCH INTERZONAL-SEARCH)
```

Next AUTOGNOSTICONROUTER proceeds to accomplish the *search* task. Because no path was retrieved from ROUTER's memory, AUTOGNOSTICONROUTER evaluates the *case-based* method to be inapplicable in the current situation. Thus, it invokes the *model-based* method which introduces at the top of its agenda the *intrazonal-search* and *interzonal-search* tasks.

```
Autognostic:    Task to accomplish INTRAZONAL-SEARCH
Autognostic:    Accomplishing Task INTRAZONAL-SEARCH
Autognostic:    Potentially applicable methods: (BFS-SEARCH-METHOD)
Autognostic:    Applicable methods: (BFS-SEARCH-METHOD)
Autognostic:    The task INTRAZONAL-SEARCH can be accomplished
     by method BFS-SEARCH-METHOD
Autognostic:    This method decomposes the task into the subtasks
     (INIT-SEARCH PATH-INCREASE)
```

AUTOGNOSTICONROUTER evaluates the conditions under which the *intrazonal-search* task is meaningful. In the current context the *final-point* also belongs in the *initial-zone*, the neighborhood in which the *initial-point* belongs. Thus the task at hand should indeed be accomplished. There is only one method applicable for accomplishing the *intrazonal-search* task, that is the *bfs-search-method* method. This method sets up as the next subtasks of AUTOGNOSTICONROUTER the tasks *init-search* and a repeated sequence of the *path-increase* task, until a *desired-path* has been produced.

```
Autognostic:    Task to accomplish INIT-SEARCH
Autognostic:    Accomplishing Task INIT-SEARCH
     by invoking the procedure INITIALIZE-SEARCH

     Router:    initialize search, with input (10th center)
     Router:    output (((10th center)))
```

The invocation of the *initialize-search* procedure returns a value for the information *possible-paths*. AUTOGNOSTICONROUTER introduces this value in the current information context, and proceeds to accomplish its next subtask, *path-increase*.

```
Autognostic:    Task to accomplish PATH-INCREASE
Autognostic:    Accomplishing Task PATH-INCREASE

New Information context for the instantiation of the STEP-IN-PATH-INCREASE
{ (cnnt-paths (((10th center)))) }

Autognostic:    Task to accomplish STEP-IN-PATH-INCREASE
Autognostic:    Accomplishing Task STEP-IN-PATH-INCREASE
Autognostic:    Potentially applicable methods: (PATH-INCREASE-METHOD)
Autognostic:    The task STEP-IN-PATH-INCREASE can be accomplished
     by method PATH-INCREASE-METHOD
Autognostic:    This method decomposes the task into the subtasks
     (GET-CRNT-PATH EXPAND-CRNT-PATH SELECT-DESIRED-PATH COMBINE-PATH-LISTS)
Autognostic:    Task to accomplish GET-CRNT-PATH
Autognostic:    Accomplishing Task GET-CRNT-PATH
     by invoking the procedure GET-CRNT-PATH

     Router:    poping the first potential path for expansion
```

```
     Router:   output  ((10th  center))

Autognostic:     Task  to  accomplish   EXPAND-CRNT-PATH
Autognostic:     Accomplishing   Task  EXPAND-CRNT-PATH
     by  invoking  the  procedure   EXPAND-CRNT-PATH
Autognostic:     Task  to  accomplish   SELECT-DESIRED-PATH
Autognostic:     Accomplishing   Task  SELECT-DESIRED-PATH
     by  invoking  the  procedure   PICK-PATH
Autognostic:     Task  to  accomplish   COMBINE-PATH-LISTS
Autognostic:     Accomplishing   Task  COMBINE-PATH-LISTS
     by  invoking  the  procedure   COMBINE-PATH-LISTS
.........
Autognostic:     Task  to  accomplish   PATH-INCREASE
Autognostic:     Accomplishing   Task  PATH-INCREASE

New  Information   context  for  the  instantiation   of  the  STEP-IN-PATH-INCREASE
{ (crnt-paths   ((((10th  center)(10th   atlantic)(atlantic     ferst-1))
             ((10th  center)(10th   techwood)(techwood     north))
             ((10th  center)(10th   hemphill  mcmillan)(hemphill     ferst-1))))   }

Autognostic:     Task  to  accomplish   STEP-IN-PATH-INCREASE
Autognostic:     Accomplishing   Task  STEP-IN-PATH-INCREASE
Autognostic:     Potentially   applicable   methods:    (PATH-INCREASE-METHOD)
Autognostic:     The task  STEP-IN-PATH-INCREASE     can  be  accomplished
     by  method  PATH-INCREASE-METHOD
Autognostic:     This  method  decomposes   the  task  into  the  subtasks
     (GET-CRNT-PATH    EXPAND-CRNT-PATH    SELECT-DESIRED-PATH     COMBINE-PATH-LISTS)
Autognostic:     Task  to  accomplish   GET-CRNT-PATH
Autognostic:     Accomplishing   Task  GET-CRNT-PATH
     by  invoking  the  procedure   GET-CRNT-PATH

     Router:   poping  the  first  potential   path  for  expansion
     Router:   output  ((10th  center)(10th   atlantic)(atlantic     ferst-1))

Autognostic:     Task  to  accomplish   EXPAND-CRNT-PATH
Autognostic:     Accomplishing   Task  EXPAND-CRNT-PATH
     by  invoking  the  procedure   EXPAND-CRNT-PATH
Autognostic:     Task  to  accomplish   SELECT-DESIRED-PATH
Autognostic:     Accomplishing   Task  SELECT-DESIRED-PATH
     by  invoking  the  procedure   PICK-PATH
Autognostic:     Evaluating   the  behavior   of  SELECT-DESIRED-PATH
     (SAME-POINT
        FINAL-NODE
          (DESIRED-PATH:=
          ((10th  center)   (10th  atlantic)(atlantic     ferst-1)(ferst-1     dalney)))
        FINAL-POINT:=(ferst-1     dalney))
Autognostic:     Task  SELECT-DESIRED-PATH     successfully    completed
```

The segment trace above illustrates the repeated execution of the *path-increase* task. In fact, for reasons of compactness, it illustrates only the first and last step in this cycle. The task *path-increase* does not have associated with it any methods, to decompose it in simpler subtasks, or procedures, to directly solve it. Instead, in its specification in the SBF model it is characterized as an instance of the *step-in-path-increase* task. Thus when AUTOGNOSTICONROUTER proceeds to accomplish it, it creates a new information context and proceeds to accomplish this prototype task in this new context.

Let us now discuss how AUTOGNOSTICONROUTER instantiates a prototype task. AUTOGNOSTICONROUTER establishes a new information context by selecting these types of information in the current context which are relevant to the accomplishment of the prototype task. At this point in AUTOGNOSTICONROUTER's reasoning, the information context contains the information { (initial-point (10th center)) (final-point (ferst-1 dalney)) (top-neighborhood z1) (initial-zone z1) (final-zone z1) (possible-paths (((10th center)))) }, not all of which is relevant to the accomplishment of the prototype task. The prototype task step-in-path-increase requires as input the information { crnt-paths destination a-zone } which corresponds to the input of its instance path-increase, { possible-paths final-point initial-zone }. Thus AUTOGNOSTICONROUTER sets up a "private" information context for the accomplishment of the prototype which contains the information shown in the top of the above trace segment. When the prototype task has been accomplished, AUTOGNOSTICONROUTER exports the information pertinent to the calling context. This information is essentially the output desired of the path-increase task, i.e., the updated possible-paths and the desired-path, which correspond to the output new-crnt-paths and path of its prototype.

This encapsulation of information around the tasks to which they are relevant enables AUTOGNOSTICONROUTER to organize its record of the problem-solving process in a way that enables the efficient discovery of new semantic relations, in case it later needs to perform a task-structure adaptation. One important issue in concept discovery is the definition of the space of hypotheses in which the learner searches for the desired concept. This space is defined by two dimensions: first the syntax in which the desired concept is to be described, (i.e., the expressiveness of the language for the target hypothesis), and second the set of attributes in terms of which the concept will be defined. AUTOGNOSTICONROUTER's information encapsulation mechanism provides an initial account for selecting the set of attributes in terms of which new functional semantics can be described. For example, if at some later point, AUTOGNOSTICONROUTER recognized the need for refining or extending the semantics of a task in the step-in-path-increase task structure, it would consider as possible terms only the types of information in the private information context of this task. This approach has the additional benefit, that modifications to the step-in-path-increase task structure, which were made as this task was employed in service of the particular route-planning task, could, in principle, be transferable even if the step-in-path-increase task was employed in the service of a different route-planning task. For a more extended analysis on attribute selection, see Section 7.7.

After each repetition of the path-increase subtask, AUTOGNOSTICONROUTER evaluates the exit condition of the cycle. As long as this condition is not met, it proceeds to spawn another path-increase task. In each one of these cycles, AUTOGNOSTICONROUTER performs four tasks. First, it selects a path, crnt-path, from its pool of paths that it can potentially expand, possible-paths (this is the get-crnt-path subtask). Next, it expands this path with all the intersections neighboring its final node, thus producing a new set of paths called expanded-paths (this is the expand-crnt-path subtask). Then, it inspects the set of expanded-paths in order to identify whether any of them reaches its destination, destination (this is the select-desired-path subtask). If none of the expanded paths is the desired one, then it proceeds to append the expanded-paths to its crnt-paths from which it has excluded the crnt-path. This new set will constitute the input crnt-paths of the next repetition.

At the beginning of each subsequent repetition of the path-increase task, AUTOGNOSTICONROUTER also evaluates whether or not the information context in which the new repetition is going to be performed is different than the information contexts of all the previous cycles. When a method sets up a repetition of one of its subtasks, all the repetitions are performed in service of the same task, i.e., the task to which the method is applied. In this example, this task is the intrazonal-search task. If there is a repetition with the exact same information context as an earlier one, then the exact same task is spawned twice in the service of the same super-ordinate task; this is a definite sign of no progress, and AUTOGNOSTICONROUTER recognizes it as a failure. A similar situation occurs when an instantiation of task is spawned in service of the same task.

Note that AUTOGNOSTICONROUTER's ability to recognize these kinds of failures lies in its knowledge of the compositional semantics of the problem-solver's task structure. The problem-

solver's methods, as specified in the SBF model, capture in service of which task each particular subtask is spawned. This knowledge enables AUTOGNOSTICONROUTER to recognize that a subtask has not fulfilled its role, if a subsequent task is spawned for the exact same role.

In the second part of the above segment, the last repetition in this episode begins with the set { (crnt-paths (((10th center) (10th atlantic) (atlantic ferst-1)) ((10th center) (10th techwood) (techwood north)) ((10th center) (10th hemphill mcmillan) (hemphill ferst-1)))) } as information context. It selects as its *crnt-path* the first one, which is expanded, among other paths, to a path that reaches the *destination*. Thus AUTOGNOSTICONROUTER successfully completes the *select-desired-path* which results in satisfying the exit condition of the *path-increase* cycle.

```
Autognostic:    Evaluating the behavior of INTRAZONAL-SEARCH
     (FORALL n IN
        NODES
          (DESIRED-PATH:=
          ((10th center)(10th atlantic)(atlantic ferst-1)(ferst-1 dalney)))
     (ZONE-INTERSECTION INITIAL-ZONE:=z1 n))
Autognostic:    Task INTRAZONAL-SEARCH successfully completed
Autognostic>   Task INTERZONAL-SEARCH
Autognostic:    This task does not contribute any information in the current context
Autognostic:    Task SEARCH successfully completed
Autognostic:    Task to accomplish STORE
Autognostic:    Accomplishing Task STORE
     by invoking the procedure STORE-CASE
Autognostic:    Evaluating the behavior of ROUTE-PLANNING
     (SAME-POINT
        INITIAL-NODE
          (DESIRED-PATH:=
          ((10th center)(10th atlantic)(atlantic ferst-1)(ferst-1 dalney)))
        INITIAL-POINT:=(10th center))
     (SAME-POINT
        FINAL-NODE
          (DESIRED-PATH:=
          ((10th center)(10th atlantic)(atlantic ferst-1)(ferst-1 dalney)))
        FINAL-POINT:=(ferst-1 dalney))
Autognostic:    Task ROUTE-PLANNING successfully completed
The value of the DESIRED-PATH is
     ((10th center)(10th atlantic)(atlantic ferst-1)(ferst-1 dalney))
```

At this point, AUTOGNOSTICONROUTER recognizes that it has accomplished not only the last cycle of the *path-increase* task but also its superordinate task *intrazonal-search*, and the super-ordinate task of *intrazonal-search* as well, *search* (since the other subtask of *search*, *interzonal-search*, does not contribute in the current episode). There are two types of knowledge that enable AUTOGNOSTICONROUTER to establish the successful completion of a higher-level, non-leaf, subtask: first, the compositional semantics of this task's decomposition by the invoked method, and second the functional semantics of the task itself. When all the subtasks of a higher-level task have been accomplished, AUTOGNOSTICONROUTER evaluates the functional semantics of this task. As in the case of leaf tasks, if the task semantics are satisfied by the actual values of the information produced as output by this task, AUTOGNOSTICONROUTER infers this task has been successfully completed. Notice, that establishing the success of a higher-level task evaluates a larger segment of the problem-solver's reasoning, as opposed to establishing the success of a leaf task which evaluates a single inference. Thus, based on its understanding of the task-structure decomposition, AUTOGNOSTICONROUTER is able to evaluate the progress of its reasoning at several levels of granularity.

At this point, AUTOGNOSTICONROUTER proceeds to accomplish its final subtask, store . Finally, it reports that it has successfully completed the route-planning task assigned to it, and the desired-path is *((10th center) (10th atlantic) (atlantic ferst-1) (ferst-1 dalney))*.

Throughout its reasoning on the particular problem, AUTOGNOSTICONROUTER has kept a record reflecting all the information shown in the trace above, i.e., which tasks were spawned, which of them were accomplished, in which order, in which manner (procedure call, method invocation, prototype instantiation), and what information they produced. All the elements involved in the problem-solving process are mentioned in this record. This record will provide the space in which the subsequent blame-assignment process will search for the causes of AUTOGNOSTICONROUTER's failure.

8.2 Blame Assignment

After the seemingly successful completion of its problem-solving process, AUTOGNOSTICONROUTER receives feedback from its external environment which suggests that the produced desired-path is not acceptable. Instead, another solution should have been produced, which is also given to AUTOGNOSTICONROUTER as part of the feedback.

```
(assign-blame-for-suboptimal-solution
                    type of information := desired-path
                    preferred solution := ((10th center)(10th dalney)(ferst-1 dalney)))
```

The role of blame assignment in the context of a failure-driven learning process is to identify a (set of) possible cause(s) for the problem-solver's failure. In addition, the identified causes should be expressed in a language such that, the subsequent repair task can easily operationalize them into adaptations to the problem solver.

There are two important issues that arise in the development of a method for the general blame-assignment task:

1. one pertains to the class of causes of failures that this method is able to identify in the problem solver, and
2. the other pertains to its efficiency.

The taxonomy of causes of failures that the blame-assignment process can identify defines the space of adaptations that the overall learning process can perform to the problem solver. In turn, the range of possible adaptations correlates with the types of performance improvement that learning is able to bring about in the problem-solver's performance. Consequently, the repertoire of the causes of failures that blame assignment is capable of identifying plays a critical role in the ability of the agent to improve its own performance. An important dimension along which the types of causes of problem-solving failures can be categorized, is whether they identify errors in the content or the organization of the problem-solver's domain knowledge or in the content or the organization of its functional architecture. In principle, a method for the general blame-assignment task should be able to address both these types, since in a realistic scenario, the state of the agent's external environment may change, (thus giving rise to the need for domain-knowledge adaptations), and the types of problems the agent is called to solve may vary, (thus giving rise to the need for adaptations to its functional architecture).

The second important issue in developing a method for blame assignment is its cost. Even though the agent may have a record of its own problem solving, which specifies all the problem-solver's elements that were involved in the failed episode, in cases of complex problem solvers

reasoning on complex problems the trace may be extremely large. As a result, if the blame-assignment method has to inspect the whole trace its cost may increase excessively. Therefore, the blame-assignment method should be able to acquit large segments of the trace and focus as fast as possible to these elements of the problem solver that are responsible for its failure. Let us now see how AUTOGNOSTICONROUTER's blame-assignment method addresses these issues in our example.

```
Autognostic:     assigning  blame  for  the  value
     ((10th  center)(10th   atlantic)(atlantic     ferst-1)(ferst-1     dalney)),
Autognostic:     instead  of  the  value
     ((10th  center)(10th   dalney)(ferst-1     dalney)),
Autognostic:     of  information   DESIRED-PATH
Autognostic:     at  the  level  of  task  ROUTE-PLANNING
Autognostic:     testing  whether  the  semantic  relations
     (SAME-POINT
           INITIAL-NODE
             (DESIRED-PATH:=
             ((10th  center)(10th   dalney)(ferst-1     dalney)))
           INITIAL-POINT:=(10th      center))
     (SAME-POINT
           FINAL-NODE
             (DESIRED-PATH:=
             ((10th  center)(10th   dalney)(ferst-1     dalney)))
           FINAL-POINT:=(ferst-1      dalney))
of  task  ROUTE-PLANNING    are  upheld  by  the  desired  solution

At  this  level  of  the  task  structure,   no  errors  have  been  identified.
Thus  AUTOGNOSTIC   expands  the  scope  of  the  investigation   into  finer  level
analysis   of  the  task  structure   by  expanding   the  task  under  investigation
     ROUTE-PLANNING
into  its  constituent   subtasks,   (SEARCH),
as  resulting   by  ROUTE-PLANNING-METHOD
```

First, AUTOGNOSTICONROUTER evaluates whether the behavior desired of the problem solver is within the capabilities of the task structure it was designed with. To do that, AUTOGNOSTICONROUTER evaluates whether or not the preferred solution satisfies the functional semantics of its overall task. In the particular example, this test amounts to evaluating whether or not the preferred path begins at the *initial-point* and ends at the *final-point*. Because these semantics are verified by the preferred solution, AUTOGNOSTICONROUTER establishes that, in principle, its task structure should be able to produce the preferred solution.

Notice that AUTOGNOSTICONROUTER's ability to decide whether or not it is at all "reasonable" to expect the preferred solution from the problem solver, relies on its understanding of the functional semantics of the problem-solver's tasks. While monitoring, AUTOGNOSTICONROUTER evaluates the actual behavior against its design specifications, that is, it evaluates whether what the problem solver does conforms with what it was designed to do. In a similar manner, during blame assignment AUTOGNOSTICONROUTER evaluates the behavior desired of it against its design specifications, that is, it evaluates whether or not the alternative behavior desired of the problem solver is within the capabilities it was designed with.

Given that AUTOGNOSTICONROUTER has established that *route-planning* could have produced the preferred solution, it infers that the cause of its failure to produce it must lie within the process that accomplishes the overall *route-planning* task. As a result, if focuses on the subtask of *route-planning* which produces as output the information *desired-path*, *search*.

To localize its search, the blame-assignment method uses the trace of the problem solving to identify which method was actually used to accomplish the task currently under investigation. Next,

based on the SBF model of the problem solver and the trace, it identifies the last subtask of this method which was actually accomplished during the failed problem-solving episode that produces the feedback information. This will be the task on which the blame-assignment method will focus next.

```
Autognostic:    assigning  blame  for  the  value
     ((10th  center)(10th   atlantic)(atlantic     ferst-1)(ferst-1     dalney)),
Autognostic:    instead  of  the  value
     ((10th  center)(10th   dalney)(ferst-1     dalney)),
Autognostic:    of  information   DESIRED-PATH
Autognostic:    at  the  level  of  task  SEARCH

At  this  level  of  the  task  structure,   no  errors  have  been  identified.
Thus  AUTOGNOSTIC   expands  the  scope  of  the  investigation   into  finer  level
analysis   of  the  task  structure   by  expanding   the  task  under  investigation
     SEARCH
into  its  constituent   subtasks,   (INTRAZONAL-SEARCH),
as  resulting   by  INTRAZONAL-SEARCH-METHOD


Autognostic:    assigning  blame  for  the  value
     ((10th  center)(10th   atlantic)(atlantic     ferst-1)(ferst-1     dalney)),
Autognostic:    instead  of  the  value
     ((10th  center)(10th   dalney)(ferst-1     dalney)),
Autognostic:    of  information   DESIRED-PATH
Autognostic:    at  the  level  of  task  INTRAZONAL-SEARCH
Autognostic:    testing  whether  the  semantic  relations
     (FOR-ALL   n  IN
           NODES
             (DESIRED-PATH:=
             ((10th  center)(10th   dalney)(ferst-1     dalney))
           (INVERSE   ZONE-INTERSECTIONS)     n  INITIAL-ZONE:=z1)
of  task  INTRAZONAL-SEARCH     are  upheld  by  the  desired  solution
Violation   of  task  semantics   by  the  desired  solution
⇒  Inferring   alternative   values  for  information-type     INITIAL-ZONE
    Alternative   value  of  INITIAL-ZONE   is  za
```

AUTOGNOSTICONROUTER keeps on refining the focus of its blame assignment as long as the tasks it investigates could have produced the solution desired of the problem solver. Thus in this example, it recursively refines the scope of the blame-assignment task from *route-planning*, to *search* and subsequently, *intrazonal-search*.

At the level of the *intrazonal-search* task, AUTOGNOSTICONROUTER recognizes that the behavior desired of this task, i.e., the transformation of its input to the preferred path, does not meet its design specifications. The preferred solution could not have been produced by this task, given the input of the task during the failed problem-solving episode, and the nature of the transformation it performs. However, AUTOGNOSTICONROUTER notices that not all the input information consumed by the *intrazonal-search* task is part of the initial information context presented to AUTOGNOSTICONROUTER from its environment. In fact, this task consumes some information which is produced by the problem solver itself, namely the neighborhood *initial-zone* in which the search is performed. Under these conditions, AUTOGNOSTICONROUTER infers that the reason that *intrazonal-search* did not produce the preferred solution may be because its input information was not appropriate. To postulate an alternative input which could potential allow the production of the preferred solution, AUTOGNOSTICONROUTER searches the domain of *zones* in ROUTER's knowledge, and uses the functional semantics of the task to select one value in this

domain. In the particular example, AUTOGNOSTICONROUTER evaluates whether any of ROUTER's neighborhoods contains all the intersections mentioned in the preferred path. Indeed there is such a neighborhood, *za*. At this point, the focus of the blame-assignment process moves from identifying why the desired output path was not produced, to why the desired initial-zone was not produced.

In this segment of AUTOGNOSTICONROUTER's behavior, yet another use of AUTOGNOSTICONROUTER's knowledge about the functional semantics of the problem-solver's tasks becomes evident. The task semantics constitute an abstract characterization of the nature of the relations that hold true between the task's input and output. As such, they can potentially enable AUTOGNOSTICONROUTER to regress the value of the output desired of a task, and to postulate the input which could lead that task to produce this output. Thus, to some extent, AUTOGNOSTICONROUTER can infer what the desired problem-solving behavior should have been, based on the outcome of this behavior, i.e., the preferred solution, without actually exploring its problem space. This is unlike Prodigy's solution which, when it fails, explores the problem space for an alternative solution, sometimes exhaustively. For a more detailed discussion on how AUTOGNOSTICONROUTER is able to infer alternative desired values for types of information produced while problem solving, see Section 6.6.2.

The main advantage of this approach is that it is applicable even when the actual problem-solving process is incapable of actually producing the desired behavior. Consider for example AUTOGNOSTICONROUTER's inference regarding what the appropriate initial-zone should have been; it does not depend on whether or not this value can actually be produced by the task responsible for producing it, i.e., classification. It may well be that the current design and/or implementation of the classification task does not allow the production of *za* as initial-zone in this problem scenario. In such case, the approach adopted in Prodigy, i.e., exhaustive problem solving, would have failed to infer the desired initial-zone value. Such a failure would imply the failure of the overall learning method since this learning method relies on the availability of the trace of the production of the preferred solution. In the case of AUTOGNOSTICONROUTER, irrespective of how classification works, AUTOGNOSTICONROUTER is able to infer the desired value for the information initial-zone based on the semantics of the task that consumes it. Furthermore, in case that this value is in conflict with the design of the classification task, AUTOGNOSTICONROUTER has a basis for postulating that the role of classification in the context of the overall route-planning process is ill-designed.

```
Autognostic:    assigning  blame  for  the  value
     z1,
Autognostic:    instead  of  the  value
     za,
Autognostic:    of  information   INITIAL-ZONE
Autognostic:    at  the  level  of  task  CLASSIFICATION
Autognostic:    testing  whether  the  semantic  relations
     (ZONE-INTERSECTIONS     INITIAL-ZONE:=za     INITIAL-POINT:=(10th     center))
of  task  CLASSIFICATION    are  upheld  by  the  desired  solution

At  this  level  of  the  task  structure,   no  errors  have  been  identified.
AUTOGNOSTIC   cannot  further  refine  the  scope  of  the  investigation
the  task  CLASSIFICATION    is  a  leaf  task
⇒ The  set  of  potential   causes  for  the  failure  are:

          Potential   Cause:   UNDER-CONSTRAINED-TASK-FUNCTIONALITY
          of  task:   CLASSIFICATION
          Desired  Behavior   in  this  Example:    INITIAL-ZONE   should  be  ZA  instead  of  Z1

          Potential   Cause:   INCORRECT-DOMAIN-RELATION-ORGANIZATIO          N
          Relation   ZONE-INTERSECTIONS     should  be  FALSE
          for  the  values  (INITIAL-ZONE    Z1)  and  (INITIAL-POINT     (10TH  CENTER))
```

Having inferred a preferred value for initial-zone AUTOGNOSTICONROUTER proceeds to assign blame for the non-production of this preferred initial-zone at the level of the classification task, which is responsible for producing it in the task structure of route-planning .

Notice that at this point, AUTOGNOSTICONROUTER has acquited the internal workings of the process that accomplishes the intrazonal-search task. Thus, it can ignore a very big segment of the trace of problem solving. AUTOGNOSTICONROUTER's ability to acquit a set of problem-solving elements relies partly on its ability to shift the assignment of blame from one type of information to another. As soon as AUTOGNOSTICONROUTER established that the task intrazonal-search could produce the desired-path , if only it was given the right input initial-zone , it has established that this task is not at fault. Therefore, neither can be the subtasks that give rise to its accomplishment. Based on its comprehension of the compositional semantics of the task structure, AUTOGNOSTICONROUTER acquits not only intrazonal-search but also the complete task-structure below this task. This enables the efficient localization of the assignment of blame.

By inspecting the functional semantics of the classification task, AUTOGNOSTICONROUTER notices that the behavior desired of this task is indeed within its design specification. Since this task is a leaf task, AUTOGNOSTICONROUTER cannot focus any further its investigation for the cause of the failure. Thus it concludes with the following hypotheses regarding the potential cause of the failure: either the functionality of the classification task is under-specified and thus it allows the production of both desired and undesired inferences, or the domain knowledge on which this task relies, in order to draw its inferences, is wrong and therefore sometimes it produces undesired values for the information initial-zone .

It is interesting to notice the dual role of the task semantics at this step. First, by explicitly specifying a concept for what the right class of inferences should be for a given task, it enables AUTOGNOSTICONROUTER to recognize that the current specification of this concept includes false positives, and therefore the concept should be refined. The ability to recognize an overly general (or too restrictive) specification of a functional concept is equivalent to recognizing errors in the problem-solver's functional architecture, which is one of the two major types of causes of failures that a blame-assignment method should be able to recognize.

On the other hand, often a functional concept may be specified in terms of the problem-solver's domain knowledge. In such cases, AUTOGNOSTICONROUTER can postulate a link from the failure of the problem solver to draw the desired inference, to potential incorrectness or incompleteness of the domain knowledge that defines the concept characterizing this class of inferences. For example, in the case of the problem-solving-and-learning scenario we are discussing, the functional concept characterizing the class of classification inferences is defined in terms of a domain relation, i.e., zone-intersections . This fact establishes a link between the inferences of the classification task and the domain knowledge on which they depend. This is how AUTOGNOSTICONROUTER can postulate as a potential cause of the failure the erroneous organization of this domain relation. And this is the second of the two major type of causes of failures that a blame-assignment method should be able to recognize, namely domain-knowledge errors.

8.3 Repair

The role of the repair task in the context of the overall failure-driven learning process is to select and eliminate one of the causes identified by the blame-assignment task, so that the problem solver does not fail again for the same reason in the future.

repair potential causes := under-constrained-task-functionality(classification)
incorrect-domain-organization(zone-intersections)

The major challenge that the agent phases in the accomplishment of this task is that it has to modify the problem solver, while, at the same time, keeping it consistent. This is especially difficult when the modification is a modification to the problem-solver's functional architecture. Let us go back to our problem scenario and discuss AUTOGNOSTICONROUTER's approach to this problem.

8.3.1 Selecting Which Cause to Eliminate

```
      Cause Selected to be addressed:    UNDER-CONSTRAINED-TASK-FUNCTIONALITY
      To task:  CLASSIFICATION
      Desired Behavior in this Example:  INITIAL-ZONE should be ZA instead of Z1

Autognostic:    Modifying according to that cause.
```

At this point, AUTOGNOSTICONROUTER has identified two potential causes for its failure to produce the desired path: one implies the revision of the functionality of a task in the problem-solver's functional architecture, and the other implies the reorganization of a domain relation. AUTOGNOSTICONROUTER prefers to attempt to eliminate the hardest cause first. The rationale behind this decision is that if this is indeed causing the problem-solver's failure, then the elimination of any other secondary cause will not eliminate the problem. Thus, in this scenario, it decides to attempt first to discover what the right functionality of the classification task must be. Because, if indeed this task is ill-designed, then it will continue producing wrong inferences even if the domain relation it is based on is locally reorganized as the competing potential modification suggests. For a more detailed discussion on how AUTOGNOSTICONROUTER selects which among the potential causes of its failure to address, see Section 7.1.

8.3.2 Selecting Which Repair Plan to Apply

```
Autognostic:    First, I'll try to identify an existing task,
     with which I can substitute the failing one, and
     get the following desired behavior for this example.
     Input:  (INITIAL-POINT  (10th center))
             (FINAL-POINT  (ferst-1  dalney))
             (TOP-NEIGHBORHOOD  z1))
     Output:  (INITIAL-ZONE  za)
Autognostic:    there is no task which I can use as a substitute for the failing one.
     Thus, I will try to postulate the semantics
     for appropriately describing the functionality of the failing task.
Autognostic:    The only option for a discriminating relation is the following:

     FUNCTION  (INVERSE  CHILDREN-ZONES)
     QUALIFIER  (FOR-ALL  INFO2)
     INFO1  INITIAL-ZONE
     INFO2  INTERMEDIATE-INITIAL-ZONE
```

The next step is to select which plan to invoke in order to accomplish the chosen learning task. As shown in Table 7.1, there are three repair plans applicable to the learning task at hand: task substitution, task modification, and insertion of a selection task. The less costly of the three is the first one. However, it assumes the existence of a task similar to the one whose functionality needs to be refined. This knowledge condition is not met in the current scenario, i.e., there is no other task known to AUTOGNOSTICONROUTER accomplishing a variant of the failing classification task. Therefore one of the other two plans has to be invoked.

Both these other plans involve the discovery of a functional concept which will be used either to modify the task semantics of the existing *classification* task (in the case of the task-modification plan), or to specify the functionality of a selection task which will be later inserted in the *route-planning* task structure (in the case of the insert-selection task plan). Thus, at this point AUTOGNOSTICONROUTER attempts to discover such a relation.

8.3.3 Repairing the Failing Problem Solver

AUTOGNOSTICONROUTER's ability to postulate new functional concepts relies on its meta-level comprehension of the problem-solver's domain. The SBF model of the problem solver includes, in addition to the functional semantics of its tasks and the compositional semantics of its task structure as a whole, a specification of what types of knowledge there are in the problem-solver's domain (for a more detailed discussion on the contents of the SBF model of a problem solver, the reader should refer to Chapter 3). When it needs to specify a new functional concept in order to revise the functionality of a failing task, AUTOGNOSTICONROUTER generates hypotheses, based on the types of knowledge which are available to the problem solver, and which are relevant to the immediate information context of the failing task. It then evaluates the hypotheses it has formulated, against the particular inferences that this concept should produce. Had AUTOGNOSTICONROUTER not have this abstract comprehension of the problem-solver's domain knowledge, it would have been unable to infer new uses for it. Non-reflective problem-solvers do not have such abstract meta-knowledge about their own knowledge; they are able to use their knowledge through the set of inferences they are designed to draw, but they do not have any means for discovering new uses for it. For a more detailed discussion on discovering functional concepts, see Section 7.7.

```
Autognostic:    there is a relation can be used to refine
   the functionality of the failing task.
Autognostic:    There are two repair plans applicable to causes of
     the type UNDER-CONSTRAINED-TASK-FUNCTIONALITY
     when a functional concept that refines the functionality
        of the failing task has been discovered
        (a) Task modification
        (b) Insertion of a Selection Task
Autognostic:    because the task produces several types of
   information as output,
     I will not modify the failing task, instead,
     I will introduce a selection task after the failing one
     with the discovered relation as semantics.
Autognostic:    (i) Creating new elements to be integrated
   with the existing task structure
        (a) a new task which can be decomposed into
        the existing task and the new selection task, G6523
        (b) an intermediate type of information, INTERMEDIATE-INITIAL-ZONE,
        to hold the possible values of information INITIAL-ZONE
        (c) a new selection task which will deliberate upon
           the different possible values
        of the information INTERMEDIATE-INITIAL-ZONE, and
        which will select this value which satisfies the discovered relation
        (d) a method which will combine the existing CLASSIFICATION task,
        with the newly introduced task, SELECTION-AFTER-CLASSIFICATION,
        into the new task, G6523,
        which will replace CLASSIFICATION in the ROUTE-PLANNING task structure
        (e) Modifying the procedure CLASSIFICATION-PROC
        to produce a sequence of possible values instead of only one
Autognostic:    (ii) Maintaining the compositional constraints of the task structure
```

```
(a) Modifying the references from the existing task structure
    the CLASSIFICATION task,
to point to the new overall task G6523
(b) Modifying the output of the old task to be INTERMEDIATE-INITIAL-ZONE,
the information type holding the possible values
of information INITIAL-ZONE
(c) Modifying the attributes produced-by and input-to
of the information type INITIAL-ZONE
```

In this learning episode, AUTOGNOSTICONROUTER discovers a relation that can be used to characterize the desired refinement on the functionality of the classification task. Thus, at this point the knowledge requirements of both the task-modification and the selection-task-insertion plan are met. Therefore, AUTOGNOSTICONROUTER has to decide among them. When the failing task produces more than one type of information, AUTOGNOSTICONROUTER prefers the plan of inserting a selection-task, in order to avoid disturbing the internal workings of the procedure that carries out the failing task. This is the case for the classification task, and thus AUTOGNOSTICONROUTER decides to select a new task after classification to refine its functionality. For a more detailed discussion on selecting among alternative plans, see Section 7.2.

Introducing a new element in the problem-solver's functional architecture can, in principle, give rise to many "non-local" affects. These affects can potentially compromise the overall consistency of the problem-solving process. Therefore the problem of maintaining the consistency of the problem solving, while also being able to significantly adapt its functioning, is a very important problem that AUTOGNOSTICONROUTER has to address.

AUTOGNOSTICONROUTER uses its comprehension of the task-structure compositional semantics to keep track of the affects of the introduction of a new task in the task structure, and to establish the necessary information and control interactions between the new task and the rest of the task structure. The steps involved in the introduction of a selection task in the task structure are discussed extensively in Section 7.5.4. However, let us walk through them in this problem scenario.

The first step of the plan is to introduce the new task in the locality of the failing task in the functional architecture. To that end, AUTOGNOSTICONROUTER postulates a new superordinate task, g6523 , to combine the functionalities of the failed task and the new selection task. This new superordinate task is meant as a replacement of the failed task. The specific steps for that goal are shown in the (i) section of the trace segment shown above. Having completed this first substep, AUTOGNOSTICONROUTER works on establishing the interactions between this new task and the rest of the functional architecture. There are two types of compositional information that need to be maintained in order to ensure consistency of the problem-solving process:

1. the control interactions among the new task and the rest of the problem-solver's task structure need to be established, and
2. the flow of information through the task structure needs to be modified in order to use the information produced by the newly inserted task.

In this example, the first goal is accomplished by replacing the classification task with task g6523 in the set of the route-planning-method 's subtasks. The second goal is accomplished by modifying the task structure to reflect the fact that the classification task does not produce any more the initial-zone , but rather an intermediate-initial-zone , which is then fed as input to the new selection-after-classification task which then produces the initial-zone consumed by the rest of the task structure.

8.4 Verification

At this point, AUTOGNOSTICONROUTER has completed the modification of the failing problem solver, and the next step is the evaluation of this modification through further problem solving. The problem that caused the failure, which triggered the learning process, is presented once again to AUTOGNOSTICONROUTER. The trace of its reasoning is shown below.

```
(monitor task := route-planning
        information context := ( (initial-point (10th center))
                                 (final-point (ferst-1 dalney))
                                 (top-neighborhood z1)))
```

Autognostic: Task to accomplish ROUTE-PLANNING
Autognostic: Accomplishing Task ROUTE-PLANNING
Autognostic: Potentially applicable methods: (ROUTE-PLANNING-METHOD)
Autognostic: The task ROUTE-PLANNING can be accomplished
by method ROUTE-PLANNING-METHOD
Autognostic: This method decomposes the task into the subtasks
(G6523 PATH-RETRIEVAL SEARCH STORE)
Autognostic: Task to accomplish G6523
Autognostic: Accomplishing Task G6523
Autognostic: Potentially applicable methods: (G6523-METHOD)
Autognostic: The task G6523 can be accomplished
by method G6523-METHOD
Autognostic: This method decomposes the task into the subtasks
(CLASSIFICATION SELECTION-AFTER-CLASSIFICATION)
Autognostic: Task to accomplish CLASSIFICATION
Autognostic: Accomplishing Task CLASSIFICATION
by invoking the procedure CLASSIFICATION-PROC2

Router: classifying the intersections in the neighborhood hierarchy
Router: output ((10th center)(ferst-1 dalney))

Autognostic: Task to accomplish SELECTION-AFTER-CLASSIFICATION
Autognostic: Accomplishing Task SELECTION-AFTER-CLASSIFICATION
Autognostic: Task to accomplish PATH-RETRIEVAL
Autognostic: Accomplishing Task PATH-RETRIEVAL
Autognostic: Potentially applicable methods: (PATH-RETRIEVAL-METHOD)
Autognostic: The task PATH-RETRIEVAL can be accomplished
by method PATH-RETRIEVAL-METHOD
Autognostic: This method decomposes the task into the subtasks
(RETRIEVAL FIND-DECOMPOSITION-INTS)
Autognostic: Task to accomplish RETRIEVAL
Autognostic: Accomplishing Task RETRIEVAL
by invoking the procedure RETRIEVE-FROM-DOMAIN-BASED-ON-ZONES

Router: retrieve-path, with input,
Router: output *nil*

Autognostic: Task to accomplish FIND-DECOMPOSITION-INTS
Autognostic: Accomplishing Task FIND-DECOMPOSITION-INTS
by invoking the procedure FIND-INTS-FROM-MIDDLE-PATH
Autognostic: Task to accomplish SEARCH
Autognostic: Accomplishing Task SEARCH
Autognostic: Potentially applicable methods: (MODEL-BASED-METHOD)
Autognostic: The task SEARCH can be accomplished
by method MODEL-BASED-METHOD
Autognostic: This method decomposes the task into the subtasks
(INTRAZONAL-SEARCH INTERZONAL-SEARCH)

```
Autognostic:     Task to accomplish INTRAZONAL-SEARCH
Autognostic:     Accomplishing Task INTRAZONAL-SEARCH
Autognostic:     Potentially applicable methods: (BFS-SEARCH-METHOD)
Autognostic:     The task INTRAZONAL-SEARCH can be accomplished
     by method BFS-SEARCH-METHOD
Autognostic:     This method decomposes the task into the subtasks
     (INIT-SEARCH PATH-INCREASE)
Autognostic:     Task to accomplish INIT-SEARCH
Autognostic:     Accomplishing Task INIT-SEARCH
     by invoking the procedure INITIALIZE-SEARCH

     Router:  initialize search, with input (10th center)
     Router:  output (((10th center)))

Autognostic:     Task to accomplish PATH-INCREASE
Autognostic:     Accomplishing Task PATH-INCREASE

New Information context for the instantiation of the STEP-IN-PATH-INCREASE
{ (crnt-paths (((10th center))) }

Autognostic:     Task to accomplish STEP-IN-PATH-INCREASE
Autognostic:     Accomplishing Task STEP-IN-PATH-INCREASE
Autognostic:     Potentially applicable methods: (PATH-INCREASE-METHOD)
Autognostic:     The task STEP-IN-PATH-INCREASE can be accomplished
     by method PATH-INCREASE-METHOD
Autognostic:     This method decomposes the task into the subtasks
     (GET-CRNT-PATH EXPAND-CRNT-PATH SELECT-DESIRED-PATH COMBINE-PATH-LISTS)
Autognostic:     Task to accomplish GET-CRNT-PATH
Autognostic:     Accomplishing Task GET-CRNT-PATH
     by invoking the procedure GET-CRNT-PATH

     Router:  poping the first potential path for expansion
     Router:  ((10th center))

Autognostic:     Task to accomplish EXPAND-CRNT-PATH
Autognostic:     Accomplishing Task EXPAND-CRNT-PATH
     by invoking the procedure EXPAND-CRNT-PATH
Autognostic:     Task to accomplish SELECT-DESIRED-PATH
Autognostic:     Accomplishing Task SELECT-DESIRED-PATH
     by invoking the procedure PICK-PATH
Autognostic:     Task to accomplish COMBINE-PATH-LISTS
Autognostic:     Accomplishing Task COMBINE-PATH-LISTS
     by invoking the procedure COMBINE-PATH-LISTS
Autognostic:     Task to accomplish PATH-INCREASE
Autognostic:     Accomplishing Task PATH-INCREASE

New Information context for the instantiation of the STEP-IN-PATH-INCREASE
{ (crnt-paths (((10th center)(10th dalney))
              ((10th center)(10th techwood)))) }

Autognostic:     Task to accomplish STEP-IN-PATH-INCREASE
Autognostic:     Accomplishing Task STEP-IN-PATH-INCREASE
Autognostic:     Potentially applicable methods: (PATH-INCREASE-METHOD)
Autognostic:     The task STEP-IN-PATH-INCREASE can be accomplished
     by method PATH-INCREASE-METHOD
```

```
Autognostic:    This method decomposes the task into the subtasks
     (GET-CRNT-PATH  EXPAND-CRNT-PATH  SELECT-DESIRED-PATH  COMBINE-PATH-LISTS)
Autognostic:    Task to accomplish GET-CRNT-PATH
Autognostic:    Accomplishing Task GET-CRNT-PATH
     by invoking the procedure GET-CRNT-PATH

     Router:    poping the first potential path for expansion
     Router:    output ((10th center)(10th dalney))

Autognostic:    Task to accomplish EXPAND-CRNT-PATH
Autognostic:    Accomplishing Task EXPAND-CRNT-PATH
     by invoking the procedure EXPAND-CRNT-PATH
Autognostic:    Task to accomplish SELECT-DESIRED-PATH
Autognostic:    Accomplishing Task SELECT-DESIRED-PATH
     by invoking the procedure PICK-PATH
Autognostic:    Task to accomplish COMBINE-PATH-LISTS
Autognostic:    Accomplishing Task COMBINE-PATH-LISTS
     by invoking the procedure COMBINE-PATH-LISTS
Autognostic:    Task to accomplish STORE
Autognostic:    Accomplishing Task STORE
     by invoking the procedure STORE-CASE
The value of the DESIRED-PATH is
     ((10th center)(10th dalney)(ferst-1 dalney))
```

This trace is similar to the trace segment which was discussed in Section 8.1 above, with two exceptions: first it mentions a new functional element, namely the task *selection-after-classification*, and second, it results in the production of the desired solution. Because the preferred solution is produced by the modified problem solver the second time, AUTOGNOSTICONROUTER evaluates this reflective episode is successful and proceeds to a new problem-solving-and-learning cycle. Had not the problem solver been successful the second time around, AUTOGNOSTICONROUTER would have undone its modification and it would have resorted to the other modification suggested by the blame-assignment task, namely the reorganization of the *zone-intersections* relation.

CHAPTER IX

EVALUATION AND ANALYSIS

The theory of reflective failure-driven learning developed in this thesis was implemented in AUTOGNOSTIC, and in this implementation, it was evaluated along four dimensions:

1. Computational Feasibility
2. Generality
3. Effectiveness
4. Realism

9.1 Computational Feasibility

AUTOGNOSTIC is a fully operational computational system implementing the reflective learning method described in Chapters 2, 5, 6 and 7. AUTOGNOSTIC is a "shell": it does not solve problems in any specific domain by itself. Instead it provides a language in which a problem solver can be specified, and a mechanism that can draw inferences about this problem solver. It can monitor the problem-solving process, assign blame for its failures, and repair it appropriately. To date, three different systems have been modeled in AUTOGNOSTIC' SBF language and have been integrated with AUTOGNOSTIC: ROUTER, a path planning system, (the term AUTOGNOSTICONROUTER refers to this integration), KRITIK2, a design system (the term AUTOGNOSTICONKRITIK2 refers to this integration), and a autonomous, reactive agent implemented in the AuRA architecture (the terms REFLECS and AUTOGNOSTICINAURA refer to this integration) [1].

AutognosticOnRouter
ROUTER is a sophisticated and yet simple path planner [Goel *et al.* 1991, Goel and Callantine 1992, Goel *et al.* 1993, Goel *et al.* 1995]. It uses multiple reasoning strategies to accomplish its task, and these strategies, in turn, use several types of knowledge. Its knowledge about its domain includes a hierarchically organized model of its micro-world, and an episodic memory of previous path-planning problems and their solutions. ROUTER operates in multiple domains, however, for the purposes of its integration with AUTOGNOSTIC it was tested in one of these domains, the Georgia-Tech campus. ROUTER already has some simple learning capabilities, independent of AUTOGNOSTIC, since it extends its path memory by storing the result of each new problem-solving episode. On the other hand, its domain is quite simple since it consists of a few types of domain concepts and a few relations between them. This makes it possible to build a detailed SBF model of both ROUTER's task structure and its domain knowledge. Because of its rich and yet not overly complicated functional architecture, ROUTER allows experimentation with several interesting scenarios that call for different learning tasks.

[1] Paul Rowland and Khaled Ali are to be credited in part for the implementation of this last integration experiment.

AutognosticOnKritik2
KRITIK2 is an adaptive design system [Bhatta and Goel 1992], [Goel 1989], [Stroulia *et al.* 1992]. Although it performs a task quite different from ROUTER, KRITIK2 too has multiple types of knowledge about its domain, including an episodic memory of cases of known devices, models which explain the functioning of these devices in terms of deep functional and causal knowledge, and semantic knowledge of the substances and the components that are available in its technological environment. KRITIK2 uses a model-based method for the overall adaptive-design task it performs, however, it has several methods for modifying the known devices to accomplish new functions; thus, KRITIK2's task structure, too, is non-deterministic. Finally, a very interesting aspect of KRITIK2 as an experimental test-bed for AUTOGNOSTIC is the fact that it uses SBF models for capturing how the devices it knows about work. These SBF models are based on a different ontology (components, substances, and structural and functional relations among them) than AUTOGNOSTIC's SBF models (tasks, methods, and knowledge) since they describe different artifacts (physical devices instead of problem solvers) but they share basically the same representation and organization scheme with AUTOGNOSTIC's SBF models. So the task of integrating AUTOGNOSTIC and KRITIK2, in addition to evaluating AUTOGNOSTIC, is an interesting experiment because it enables a comparative analysis of the roles that SBF models can play in these two different domains of designed artifacts.

AutognosticInAuRA
For the AUTOGNOSTICINAURA experiment, AUTOGNOSTIC was integrated with the reactive planner designed by the Georgia-Tech team, for the purposes of the AAAI-93 robotics competition. This reactive planner was designed and developed in the context of the AuRA architecture [Arkin 1986]. The agent resulting from this integration is called REFLECS. In this architecture, the reactive planner does not have any knowledge about its environment internally represented. Its tasks are accomplished through the run-time synthesis of a set of primitive reactive behaviors. These elementary reactive behaviors are the building blocks of behavior in a reactive system. Each one of these building blocks consists of a perception schema and a motor schema: the former schema perceives some aspect of the environment, and, based on this percept, the latter schema produces some vector along which the robot should move. The overall movement of the robot is the result of the synthesis of the individual vectors produced by the behaviors which happen to be active at the time. Examples of such elementary reactive behaviors are the *move-to-goal* behavior and the *avoid-obstacle* behavior. The *move-to-goal* behavior produces a vector pushing the robot towards a specified goal. The *avoid-obstacle* behavior produces a vector repelling the robot from all obstacles near it. As a reactive system, REFLECS is fundamentally different from ROUTER and KRITIK2, both of which are knowledge-based systems. Because it is a reactive system, REFLECS can only benefit from AUTOGNOSTIC's ability to modify the system's task structure (knowledge content or organization modifications are irrelevant in this context), and this allows for a more focused study of the classes of systems and the knowledge conditions under which task-structure modifications are useful. Finally, REFLECS was originally developed in a completely different research paradigm than the paradigm from which ROUTER, KRITIK2, and also AUTOGNOSTIC have originated.

9.2 Generality of AUTOGNOSTIC's SBF Language and Reflective Learning Process

A critical dimension along which a learning theory needs to be evaluated is its generality, namely, the class of intelligent systems it applies to, and the class of learning tasks it addresses. Many learning theories apply to a specific class of systems all of which accomplish the same reasoning task. For example, theories for learning discrimination trees are relevant to classification systems only. Furthermore, a learning theory may cover different aspects of the function of learning in the context of an intelligent system. For example, it may account for how learning expands the set of problems the system can solve, or how it improves the quality of the solutions it produces, or how it improves the efficiency of the system's reasoning.

The fact that AUTOGNOSTIC was integrated with these three different systems provides evidence

for the expressiveness of AUTOGNOSTIC's SBF language for describing the functional architecture of problem-solving systems, and the generality of its reflective learning process. Note, that all these three systems were developed independently of this thesis. ROUTER and KRITIK2 were developed in the same research paradigm, which shared the methodology of task structures as a means for designing and analyzing intelligent systems. On the other hand, AuRA, the architecture in which the reactive planner of REFLECS was developed, was designed and implemented in a completely different paradigm. Furthermore, the tasks these systems perform are quite diverse. Finally, the problem-solving theories that they embody are fundamentally different: deliberative reasoning with a lot of internal knowledge in ROUTER and KRITIK2, vs. reactive, completely without internal representations in REFLECS. These facts provide some evidence in support of the generality of AUTOGNOSTIC's SBF language and the reflection process which is based on it.

What do these three different systems have in common? They all consist of a set of identifiable design elements, each of which has a well-defined functionality. The interactions among these elements and the rules of composition for how the functionality of higher-level elements gets composed by the interactions of lower-level elements are also well-defined.

The necessity that these conditions are met becomes evident when we consider the process of integrating a particular problem solver with AUTOGNOSTIC. The first step in this process is to enumerate the design elements that comprise the problem solver, and to explicitly specify their functionality in the context of its overall behavior. Consider for example, the integration of the reactive planner with AUTOGNOSTIC which led to REFLECS. The first task was to identify and enumerate its design elements. In the paradigm in which the reactive planner was developed, these design elements are its elementary behaviors, and at a lower level, the perceptual and motor schemas of which these behaviors are combined. Having identified all the design elements involved in the production of the problem-solver's behavior, the next task was to specify the function of each one of them in terms of the information they take as input, the information they produce as output and the transformation they perform between these two. If there are design elements that have not been identified during the first step, or whose function has not been specified during the second step, then AUTOGNOSTIC will be unable to reason about them, and identify their functioning as the potential cause of the overall failure of the problem solver. The third step in the integration process, is the specification of the composition of the functions of these design elements into the overall system's behavior. If the overall behavior of the problem solver emerges from the interaction of its design elements in a way that cannot be described as a well-defined composition of their respective functions, then AUTOGNOSTIC may be able to reason about the overall function of the system and recognize its failure but it will not be able to discriminate and localize the cause further into its constituent design elements. For an extended description of the method for analyzing and specifying a problem solver in terms of the SBF language, the reader should refer to Chapter 4.

The characteristics of the class of problem solvers which can be modeled in AUTOGNOSTIC's SBF language and can be integrated with AUTOGNOSTIC are summarized in the Table 9.1 below.

Going beyond the specific systems that AUTOGNOSTIC has been tested with, it is interesting to note that, the aspects of the problem-solving process that AUTOGNOSTIC's SBF language captures have been receiving increasing attention by researchers in the fields of artificial intelligence, cognitive science, knowledge-based systems and robotics, as the "right" aspects in terms of which to analyze intelligent behavior. Consequently, it would seem that a theory of performance-driven self-adaptation based on an agent's comprehension of these aspects would apply to this class of systems which are analyzable in these terms.

There is a large corpus of artificial intelligence and cognitive science research which indicates that the nature of intelligent behavior is deliberative and goal driven [Newell and Simon 1963a, Newell and Simon 1963b]. Often, intelligent agents, while reasoning about their tasks, set up intermediate goals for themselves so as to decompose their overall task into subtasks. Furthermore, when they reach some kind of impasse [Laird *et al.* 1986, Laird *et al.* 1987] intelligent agents may set up subgoals which aim to provide the right knowledge context for resolving that impasse. In addition, recent results in knowledge-based systems research [Marcques *et. al* 1992, McDermott 1981, Steels 1990, Wielinga *et al.* 1992] indicate that knowledge-level analysis of the system task, i.e., analysis, in terms of subtasks and methods and knowledge, enable the processes of acquiring knowledge from human domain experts and designing flexible, understandable and

Table 9.1: The class of problem solvers that can been modeled in terms of the SBF language and to which the reflection process implemented in AUTOGNOSTIC is applicable.

Independent of	problem-solving task problem-solving domain reactive or deliberative problem-solving behavior
Conditioned upon	the complete identification of the system's design elements the specification of the function of each design element the specification of the overall system function as the composition of the functions of these design elements

reusable systems. Finally, several recent lines of research in robotics, such as the UMPRS project for example [Lee *et al.* 1994], take a similar functional analysis view for the task of designing flexible robot behaviors. All these convergent lines of research provide additional supporting evidence that the class of systems that might benefit from AUTOGNOSTIC's reflective learning process is, indeed, quite large.

9.3 Effectiveness of the Reflective Learning Process

The next dimension along which AUTOGNOSTIC's theory of reflective performance-driven learning was evaluated was its effectiveness, that is, with respect to the extent to which learning improves the performance of the intelligent system in service of which it occurs. As mentioned also in the introduction, in general, there are three dimensions in terms of which the performance of a system can be potentially improved:

1. the quality of the solutions it produces may improve,
2. the efficiency of the problem-solving process may improve, and
3. the population of problems that the system can solve may increase.

In addition, the affects of performance-driven learning to the system's performance can be evaluated at several different levels of experience with problem solving. For example, even if there is relative improvement within a problem-solving-and-learning cycle, there is no not guarantee that, in the long run, the performance of the problem solver will improve for the complete class of problems it addresses. Thus, there is a need for a separate evaluation of the effectiveness of a learning theory after a sequence of problem-solving-and-learning episodes.

Thus, in general, the affects of learning can be evaluated

1. after a single problem-solving-and-learning episode, or
2. after a sequence of such episodes.

These two variables, i.e., dimension of performance improvement and amount of problem-solving experiences, define a set of six different conditions in which the effectiveness of AUTOGNOSTIC was evaluated. These conditions, and the systems in the context of which AUTOGNOSTIC was evaluated, are summarized in the grid of the Table 9.2 below.

The following three sections of this chapter describe and analyze the experiments conducted with each one of the systems integrated with AUTOGNOSTIC.

Table 9.2: Conditions in which the effectiveness of AUTOGNOSTIC's reflective learning was evaluated.

	Quality of Solutions	Efficiency of Problem Solving	Population of Solvable Problems
individual episode	Router Kritik2	Router Kritik2	Kritik2 Reflecs
sequence of episodes	Router	Router	Router

9.4 Experiments with AUTOGNOSTICONROUTER

AUTOGNOSTIC's SBF model of ROUTER's path planning, as used in the following experiments, is shown in Appendix 1.

9.4.1 Learning from an Individual Problem-Solving Episode

A set of six individual problems were presented to AUTOGNOSTICONROUTER, in order to evaluate its learning behavior in a variety of problem-solving scenarios. These problems were used to illustrate different aspects of AUTOGNOSTIC's process throughout this thesis.

Table 9.3: Learning from individual problem-solving episodes in AUTOGNOSTICONROUTER.

Example Problem	Sections in which it is discussed	Performance Improvement
1	5.2.1, 6.6.2, 7.5.4	Quality of Solution
2	5.2.2, 6.5, 7.6	–not applicable–
4	6.6.1	Quality of Solution
5	6.6.3, 7.4.2	Quality of Solution
6	7.4.2	Problem-Solving Efficiency
7	7.5.2	Problem-Solving Efficiency

The table 9.3[2] summarizes these examples, by pointing out for each one of these examples, the dimension along which AUTOGNOSTIC improves ROUTER's performance, and the sections in which it is discussed in detail.

[2]In problem 2, AUTOGNOSTICONROUTER simply modified the description of the semantics in the SBF model. It was simply a "model-revision" task, with no affects to problem-solving performance. Example problem 3 was from KRITIK2.

9.4.2 Incremental Learning from a Sequence of Problem-Solving Episodes

To evaluate the effectiveness of AUTOGNOSTIC's learning process after a sequence of problem-solving-and-learning episodes, two different sets of experiments were conducted with AUTOGNOSTICONROUTER. One was designed to investigate the long-term affects of reflection on the quality of the problem-solver's solutions, and the other was designed to investigate its affects on the efficiency of its process. A random sequence of 150 problems in ROUTER's domain was generated, and ROUTER, in its original condition, was presented with each one of them. Next, three different sequences of 40 problems, randomly selected from the original set of the 150 problems, were generated. Each of these sequences was presented to AUTOGNOSTICONROUTER in the context of two experiments. The first one evaluated the effectiveness of AUTOGNOSTICONROUTER's reflection process as a means of improving the quality of ROUTER's solutions. The second evaluated the effectiveness of AUTOGNOSTICONROUTER's reflection as a means for improving the efficiency of ROUTER's reasoning. In both of these sets of experiments, the class of problems that ROUTER was able to solve after its adaptation by AUTOGNOSTIC was increased with respect to the set of problems it could solve before. The design and the results of these experiments are reported in the sections 9.4.2.2 and 9.4.2.3.

Finally, an important issue arises when a learning theory is evaluated in a long-time scale, namely the issue of convergence. The convergence of AUTOGNOSTIC's learning was studied in all the above experiments, and the results are reported in section 9.4.2.6.

9.4.2.1 The Experimental Design

The goal of the first set of experiments was to evaluate AUTOGNOSTIC's ability to redesign ROUTER in such a way that the quality of the paths it produces improve. The chosen criterion of path quality was the real length of the path. Thus, for the quality-of-solution experiments, an oracle path planner was built, which given a particular problem, produced the shortest path connecting the two problem intersections.

As originally designed and developed, ROUTER does not have any metric information regarding its domain: its model of the world contains qualitative information only about the pathways in the domain and their intersections, and not about the real distances of the segments between the intersections of a particular street. Thus, in its original implementation, ROUTER produces a satisfying path between the initial and final problem locations, and there are no guarantees regarding its quality as far as real length is concerned. In the context of this experiment, AUTOGNOSTICONROUTER was endowed with knowledge about the real lengths of all the street segments in its domain knowledge. Furthermore, a new attribute was added in the abstract description of *path* in AUTOGNOSTIC's SBF model of ROUTER, i.e., *real-length* .

The goal of the second set of experiments was to evaluate AUTOGNOSTIC's ability to redesign ROUTER in such a way that the efficiency of its problem-solving process improved. The criterion used for process efficiency was the total number of repetitions of the *path-increase* task while searching the domain model. In ROUTER, an indirect measure of the same criterion is the "simplicity" of the path: the simpler the path, i.e., the smallest the number of intersections mentioned in the path, the more efficient the process. Thus, for the process-efficiency experiment, a second oracle was built, which given a particular problem, produced the simplest connecting the two problem intersections.

For each criterion, i.e., quality-of-solution effectiveness, and process-efficiency effectiveness, AUTOGNOSTICONROUTER was presented with three randomly generated sequences of 40 problems each. After each problem for which AUTOGNOSTICONROUTER did not produce the desired path, as defined by the oracle, AUTOGNOSTIC proceeded to reflect on the failed problem-solving episode and modify ROUTER's knowledge or reasoning process appropriately. Next, AUTOGNOSTICONROUTER proceeded to solve the problem again. If the correct path was not produced this second time either, AUTOGNOSTIC reflected on the failure and modified ROUTER once again. The problem-solving-and-learning cycle was repeated as long as AUTOGNOSTICONROUTER did not produce the correct path and AUTOGNOSTIC had a modification to suggest.

At the end of each 40-problem sequence, ROUTER was presented again with the complete 150-problem sequence, and the solutions produced by ROUTER for each path before and after

training were compared to evaluate effectiveness of AUTOGNOSTIC's reflective learning. The first set of experiments, on quality of solution, is discussed in section 9.4.2.2, and the second set of experiments, on process efficiency, is discussed in section 9.4.2.3.

9.4.2.2 Redesigning for "Quality of Solutions" Improvement

Table 9.4 reports the results of the quality-of-solution experiments.

Table 9.4: Results on Quality-of-Solution performance improvement.

# sequence	Number of Problems with Improved Path Quality	Number of Problems with Deteriorated Path Quality	Sign test	Paired t-test results
sequence 1	55	13	$1.3\ 10^{-7}$	0.0002 (t=3.88)
sequence 2	53	14	$8.8\ 10^{-7}$	0.0000 (t=5.17)
sequence 3	44	30	0.06	0.14 (t=1.49)

This set of experiments demonstrates AUTOGNOSTIC's capability to redesign a problem solver in a way that it produces better solutions, as long as AUTOGNOSTIC is given consistent feedback that satisfies the desired quality criterion. A non-parametric sign test revealed that the population of problems for which the quality of paths was improved was in all experiments significant above the .00001 point. However, it is important to note that it is not completely guaranteed that the path improves for each and every problem instance, as shown by the paired t-test results in the table. In some problems, the quality of the path produced for a particular problem after learning was worse than before, because different methods were used to solve the same problem before and after learning, or because different paths were retrieved and used by the case-based planning method.

Analysis of the "Quality-of-Solution" experiments

Let us discuss some interesting learning problems that arose in this set of experiments. First, it is interesting to notice, that although it was not an explicit optimization criterion, the population of problems solved by ROUTER increased after its redesign by AUTOGNOSTIC. The reason for that improvement is that as it got feedback from the world, AUTOGNOSTIC extended ROUTER's domain knowledge, and therefore it could solve more problems in its domain.

The original ROUTER was designed to produce satisfying and not optimal paths. Its neighborhood hierarchy was designed to organize its domain model so that ROUTER's search is local, and consequently efficient. This design however has a negative impact on the solution quality. In general, limiting the search space makes the search process less exhaustive, and this, in principle, is in conflict with the optimization of any quality criterion for the solution. In order to enable the production of better solutions, AUTOGNOSTIC expanded ROUTER's neighborhoods in all three quality-of-solution experiments. By extending the neighborhoods, AUTOGNOSTIC provides more information in each locality. This, as a result, enabled ROUTER, after its redesign by AUTOGNOSTIC, to solve more problems (94, 91, 78 after each of the three training 40-problem sequence correspondingly) with intrazonal search, where in its original condition, only 46 problems were solved by intrazonal search. This shift in the methods used to solve the problems accounts to a great extent for the improvement in the path quality, since with no exception, for all problems which were solved by intrazonal search after training, the paths produced after training were better (or in some cases equal) than the paths produced before.

Another reason why ROUTER's paths improved after its redesign by AUTOGNOSTIC is the modification of its method for the task path-increase. In the original condition, ROUTER

repetitively expands its current path until it reaches the final intersection. At each point in this cycle, ROUTER maintains a pool of paths that it can possibly expand, and each time it chooses the first in this list, which is one of the simplest paths, i.e., a path with the least number of segments. This selection process[3], i.e., selecting the first in the list of possible paths, was an implementation-level decision and not a design one. In all three quality-of-solution experiments, AUTOGNOSTIC modified this task in the exact same way, that is it refined its functionality to prefer the shortest of the available paths, although in each experiment the modification was invoked in a different context. Let us discuss here, how AUTOGNOSTIC modified the method in the context of the first experiment.

When AUTOGNOSTICONROUTER was presented with the problem of going from *(mcmillan 6th-1)* to *(cherry-1 ferst-0 5th)*, it produced the path *((mcmillan 6th-1) (10th hemphill mcmillan) (10th techwood) (techwood 5th) (cherry-1 ferst-0 5th))*. It was then given as feedback the path *((mcmillan 6th-1) (6th-1 ferst-2) (ferst-2 hemphill ferst-1) (ferst-1 dalney) (ferst-1 state) (ferst-1 atlantic) (plum ferst-1 ferst-0) (cherry-1 ferst-0 5th))*. The subpath *((mcmillan 6th-1) (6th-1 ferst-2) (ferst-2 hemphill ferst-1) (ferst-1 dalney) (ferst-1 state) (ferst-1 atlantic) (plum ferst-1 ferst-0))* was in its set of possible paths at the time when AUTOGNOSTICONROUTER arbitrarily selected the path *((mcmillan 6th-1) (10th hemphill mcmillan) (10th techwood) (techwood 5th))* for expansion. Thus, AUTOGNOSTICONROUTER recognized the need to refine the information-transformation function of the get-current-path task so as to prefer paths of the kind suggested by the feedback, and it postulated that the right paths to select for expansion were the shortest paths, since the path that it should have expanded was shorter than the path it actually expanded. Using that selection criterion, AUTOGNOSTICONROUTER was able to produce the correct path the second time it solved this problem.

The resulting "greedy" algorithm is not optimal. The greedy, shortest-path first method for the task path-increase does not always produce the optimal path. For example, in the case where there are two paths in the pool of paths that can be expanded, *path1 := ((a b)(b c)(c d))* and *path2 := ((u v)(v w)(w x))*, and *length(path1) < length(path2)*, this method will select *path1*. It may be however, that, in one more cycle this path will reach the destination and ROUTER will present the path *((a b)(b c)(c d)(d final))* as the solution even when *length(((a b)(b c)(c d)(d final))) > length(((u v)(v w)(w x)(x y)(y z)(z final)))*. In all such cases however, when, this greedy method failed and AUTOGNOSTIC attempted to redesign this method, it either did not find any alternative, or it found alternatives that did not lead to improvement and therefore it rejected them.

This decision transformed ROUTER's breadth-first search into a shortest-path first search, which enabled its intrazonal, model-based, search method to produce generally improved paths. In fact, there were several problems (8 out of 45 in the first experiment, 6 out of 38 in the second experiment, and 8 out of 38 in the third experiment) which were solved by intrazonal search in both conditions, for which ROUTER after its redesign produced better paths than ROUTER in the original condition, although both systems had all the information pertinent to the better path in the neighborhood they performed the search. There were few exceptions to that rule (0 out of 45 in the first experiment, 2 out of 38 in the second experiment, and 1 out of 38 in the third experiment) due to the reason discussed above. For these problems, the improvement is due only to the redesign of the method for the task path-increase .

A decision that has a major impact in the quality of the produced path, when the problem is not solved by intrazonal search, is the decomposition of the problem into subproblems. In the case-based method, the task responsible for this decision is the path-retrieval task, where in the inter-zonal, model-based method, the tasks which play that role are the find-source-common-int and find-dest-common-int tasks. In all these experiments, AUTOGNOSTIC redesigned ROUTER to prefer as intermediate intersections, int1 and int2 , those intersections which lie on the same street with the initial and final problem locations correspondingly. This heuristic, although not always correct, ensures to some extent that the chosen intermediate locations, being relatively close to the problem locations, will not lead the search far away from the optimal path. Thus for several (14 out of 21 in the first experiment (1 exception), 10 out of 21 in the second experiment (7 exceptions), and 10 out of 21 in the third experiment (11 exceptions)) problems solved by inter-zonal, model-based search in both conditions, ROUTER, after its redesign, produced a better path

[3]The reason for that, is that ROUTER extends its poll of paths by appending new possible paths at the end of the list, ans as newer paths are produced by expansion of older ones, they tend to be more complicated than the older paths

than the original ROUTER. The "correctness" of this heuristic depends greatly on the knowledge organization. When neighborhoods overlap over a large space, they may share long segments of streets and thus, two intersections on the same street may be far way, therefore rendering the heuristic ineffective. This dependency led to the high number of exceptions in the second and third experiments.

In trying to redesign ROUTER's method for path-retrieval in a similar manner, AUTOGNOSTIC postulated a series of heuristics for preferring specific paths from memory. The original ROUTER selects paths that begin in the current-problem's initial zone and end in the current-problem's final zone. In the first experiment, AUTOGNOSTIC modified the method to select paths with a small number of segments. In the second one, it modified it to select the shortest paths that connect the two intersections. In the third experiment, AUTOGNOSTIC modified the method to select the shortest path that begins on the same street as the initial problem location.

In all three experiments AUTOGNOSTIC postulated first that the right path to retrieve was the shortest path between the two neighborhoods. However in the first and second experiment it modified this heuristic in two different ways. The fact that the retrieval method was modified in different ways in the three experiments, and even multiple times in some of them, leads us to a not too-surprising hypothesis: that memory processes might be inherently more difficult to "optimize" than deliberative processes such as planning. This is because they are more flexible and as the memory contents change, the appropriate case-selection criteria may need to change.

The learning tasks that arose in the context of the three quality-of-solution experiments (and the corresponding modifications performed to ROUTER) are summarized in the list below:

1. Knowledge Acquisition: Knowledge about new streets and new intersections was integrated in ROUTER's domain knowledge.
2. Knowledge Reorganization: AUTOGNOSTIC expanded ROUTER's neighborhoods.
3. Task-Functionality Refinement:
   (a) AUTOGNOSTIC redesigned the method for accomplishing ROUTER's path-increase task into a greedy, shortest-path first search, by modifying the functionality of the get-current-path subtask.
   (b) AUTOGNOSTIC modified the method interzonal-decompose-method used in the accomplishment of ROUTER's interzonal-search task, by modifying the functionality of the int-selection task to prefer as common-int an intersection which lies on the same street with the anchor-int .
   (c) AUTOGNOSTIC modified the method for ROUTER's path-retrieval by modifying the retrieval subtask.

9.4.2.3 Redesigning for "Problem-Solving Efficiency" Improvement

The second set of experiments conducted with AUTOGNOSTICONROUTER was designed to evaluate AUTOGNOSTIC's ability to redesign ROUTER so that its process becomes more efficient.

An important issue that arose in that experiment was designing the feedback for AUTOGNOSTICONROUTER's training. What is the kind of feedback that might lead AUTOGNOSTIC to redesign ROUTER so that it becomes more efficient? The question of feedback was much simpler in the first experiment, since, when a particular kind of solution is desired of the problem solver, then the right feedback for each problem-solving episode is a solution to the particular problem that exhibits exactly the desired properties. The problem that arose in this experiment was to identify properties of the solutions indicative of inefficient problem solving. The task that contributes the most to the overall cost of ROUTER's problem-solving process, both in terms of time complexity and in terms of space complexity, is the path-increase task. This is because this task is repetitively executed, and the number of repetitions is exponential to the number of intersections contained in the path, where all other tasks in the task structure are performed only once. Furthermore, this task is repetitively executed and each time the output information it produces, i.e., the set of paths that

can be expanded, is added to its previous output. Thus, while all other tasks require a fixed amount of memory space for their output, the memory requirements of this task are proportional to the length of the path, and consequently this task is also the decisive space-complexity factor. Thus, for both time and space complexity, simple paths are indicative of efficient processing, and this was the feedback given to AUTOGNOSTICONROUTER during its training in this experiment.

Table 9.5 depicts the improvement of ROUTER's process efficiency before and after its redesign by AUTOGNOSTIC in each one of the three experiments. Again, the non-parametric sign test was significant for all experiments above the .00001 point.

Table 9.5: Results on Process-Efficiency performance improvement.

# sequence	Number of Problems with Fewer Cycles	Number of Problems with More Cycles	Sign test	Paired t-test results
sequence 1	60	10	$4\ 10^{-10}$	0.0000 (t=5.92)
sequence 2	50	22	$6.4\ 10^{-4}$	0.25 (t=1.15
sequence 3	56	15	$5.2\ 10^{-7}$	0.0000 (t=4.82)

Analysis of the "Problem-Solving-Efficiency" experiments

In these experiments too, AUTOGNOSTICONROUTER integrates new facts into ROUTER's domain knowledge. Irrespective of the particular quality of the paths that AUTOGNOSTICONROUTER receives as feedback, as long as they refer to elements in the domain that ROUTER does not already know about, AUTOGNOSTICONROUTER is able to improve ROUTER's domain knowledge. However, there was an interesting difference between the knowledge that AUTOGNOSTICONROUTER acquired in the two experiments. Although AUTOGNOSTICONROUTER was presented with the same sequences of problems, because the feedback it was given was different, the progress of its knowledge acquisition was different. In fact, because the paths given as feedback in this experiment were more abstract, they did not include as many references to domain elements as the paths given as feedback in the previous experiment. This can be seen in Figure 9.4 which depicts the distribution of the knowledge-acquisition modifications in this experiment. In this figure, the upward slope is slower than in the corresponding Figure 9.3 for the previous experiments.

In these experiments, AUTOGNOSTICONROUTER reorganized ROUTER's domain knowledge to a lesser degree than in the previous sequence of experiments. This can also be seen in Figure 9.6 which depicts the distribution of the knowledge-reorganization modifications in this set of experiments, as compared with Figure 9.5 which depicts the same distribution in the previous set. This comes as no surprise, since the hierarchical domain-model organization and the problem decomposition it supports contributes to increased efficiency, and therefore AUTOGNOSTIC did not disturb it but only slightly.

In all three process-efficiency experiments, AUTOGNOSTICONROUTER modified the method for carrying out the path-increase task. For all these experiments, AUTOGNOSTIC first tailored the process of selecting a path to expand by *selecting the shortest path*. This was because, in several cases (especially true when the distance between the two locations is small), short path coincides with simple path. However later problems caused AUTOGNOSTICONROUTER to redefine its selection criterion into *selecting that path which lies on the same street with the final intersection*. Remember in the first experiment too, there had been problems that caused AUTOGNOSTICONROUTER to "doubt" its original selection criterion. However, each time it tried to redefine it in the first experiment, either there was no concept that could be used to do that, or there were tentative alternative concepts would lead to deterioration of the path quality. Thus, the original criterion remained stable throughout the training session in the first set of experiments. In this set of experiments, AUTOGNOSTICONROUTER was able at a later time to modify its criterion and substitute it with a new one, more effective in

maximizing the current optimization criterion. In order however, to recognize the ineffectiveness of its original modification towards reducing efficiency, AUTOGNOSTICONROUTER was given some more local feedbacks than the overall desired path. Instead of the path desired as a solution to the given problem, which was also the path actually produced by the problem solver, it was given as feedback the information that this path could have been produced earlier. This was necessary because inefficiencies in processing do not always manifest themselves in undesirable solution qualities, that is, correct paths can be produced inefficiently. In such cases, more localized feedback, in terms of intermediate types of information is necessary.

Again in this set of experiments, AUTOGNOSTICONROUTER redesigned its int-selection task in a manner similar to the previous ones, only in these experiments there was no context where this heuristic failed.

It also redesigned the method for accomplishing path-retrieval. In the first and third experiments, it learned to prefer the shortest paths in terms of real length. In the second one, it learned to prefer paths that begin from the initial problem location.

In the first experiment of this set, AUTOGNOSTICONROUTER performed yet another task modification: namely, it redesigned its classification task to select the lowest neighborhood in the hierarchy, when a problem location belongs in more than one neighborhoods at the same time. This improves the process efficiency because search in a lower neighborhood implies search in a smaller neighborhood (remember, lower-level neighborhoods cover smaller spaces in greater detail) and thus, in general, fewer points visited during search.

The list below summarizes the learning tasks that arose in the process-efficiency experiments and the corresponding modifications performed to ROUTER:

1. Knowledge Acquisition: Knowledge about new streets and new intersections was integrated in ROUTER's domain knowledge.

2. Knowledge Reorganization: AUTOGNOSTIC modified the contents of ROUTER's neighborhoods to "smooth" artificially sharp boundaries between them.

3. Task-Functionality Refinement:

   (a) AUTOGNOSTIC redesigned the method accomplishing ROUTER's path-increase task into a greedy method, by modifying the functionality of its get-current-path subtask to select first the most promising path, i.e., one that lies on a common street with the destination.

   (b) AUTOGNOSTIC modified the method interzonal-decompose-method used in the accomplishment of ROUTER's interzonal-search task, by modifying the functionality of the int-selection task to prefer as common-int an intersection which lies on the same street with the anchor-int.

   (c) AUTOGNOSTIC modified the method for ROUTER's path-retrieval, by modifying its retrieval subtask to prefer paths with a small number of intersections.

   (d) AUTOGNOSTIC modified the functionality of ROUTER's classification so that it prefers the smaller, most specific neighborhoods, when an intersection belongs in more than one intersections at the same time.

9.4.2.4 A Commentary on the two Sets of Experiments

It is interesting to note that in all these experiments the same three tasks, path-increase, int-selection and path-retrieval, were subjected to modifications. Why? Is it an artifact of the problem list used to "train" AUTOGNOSTIC, or is there some aspect of ROUTER's original design that makes three tasks candidates for AUTOGNOSTIC's modifications? The modifications arose in different problems in the experiments, although the same problem lists were in both experiments, and their relative stability during AUTOGNOSTICONROUTER's training was different in the two experiments. These two facts support the latter hypothesis. In fact, these three tasks are the three basic elements of ROUTER's reasoning. When the two problem locations belong in the

same neighborhood, the path that ROUTER produces depends solely on its local search method, i.e., the method for path-increase . When the two problem locations belong in different neighborhoods, and the problem is solved by model-based interzonal-search , the path depends on the problem decomposition by the int-selection task and again on the local search method for solving the particular subproblems. Finally, when the case-based method is used to solve the problem, the path depends on the path retrieved from memory, and again on the local search method for adapting it.

9.4.2.5 "Population of Solvable Problems" Improvement

In all the above experiments, during training, as AUTOGNOSTICONROUTER "conversed" with its environment, it received new knowledge about its world. The different feedback paths contained references to new objects in the world, and AUTOGNOSTICONROUTER integrated these new references in ROUTER's world model. Due to this knowledge acquisition, in all these experiments the class of problems that ROUTER was able to solve after learning was a superset of the class of problems it was able to solve before. Table 9.6 summarizes the results on the growth-of-solvable-problems-population dimension of these experiments. It is interesting to note that the class of solvable problems increased to a greater extent in the experiments of the quality-of-solutions set because the feedback in these experiments included more detailed descriptions of the desired path which, in general, communicated more information to AUTOGNOSTIC than the simple path given as feedback in the efficiency experiments.

Table 9.6: Results on Population-of-Solvable-Problems performance improvement.

Condition	Number of Unsolved Problems (out of 150)
Before Learning	36
QoS, experiment 1	7
QoS, experiment 2	8
QoS, experiment 3	9
Efficiency, experiment 1	17
Efficiency, experiment 2	24
Efficiency, experiment 3	10

9.4.2.6 Convergence

An interesting question that arises when a theory of adaptation towards performance-improvement is evaluated in a long-term window, is whether this adaptation process converges. That is, will the design of the problem solver stabilize to a "better" design, in the long run? Or, will it continually be modified every time a new problem is presented? Figures 9.1, 9.3, and 9.5 depict the distribution of the modifications that AUTOGNOSTIC performed on ROUTER during the training sessions in the quality-of-solution experiments. Figures 9.2, 9.4, and 9.6 depict the distribution of the modifications that AUTOGNOSTIC performed on ROUTER during the training sessions in the process-efficiency experiments.

Figures 9.1 and 9.2 show that modifications of the functional architecture of the problem solver are essentially rare events, where modifications of its domain knowledge are far more often. The big "steps" in the Figure correspond to problems that caused the initial redesign of the methods for

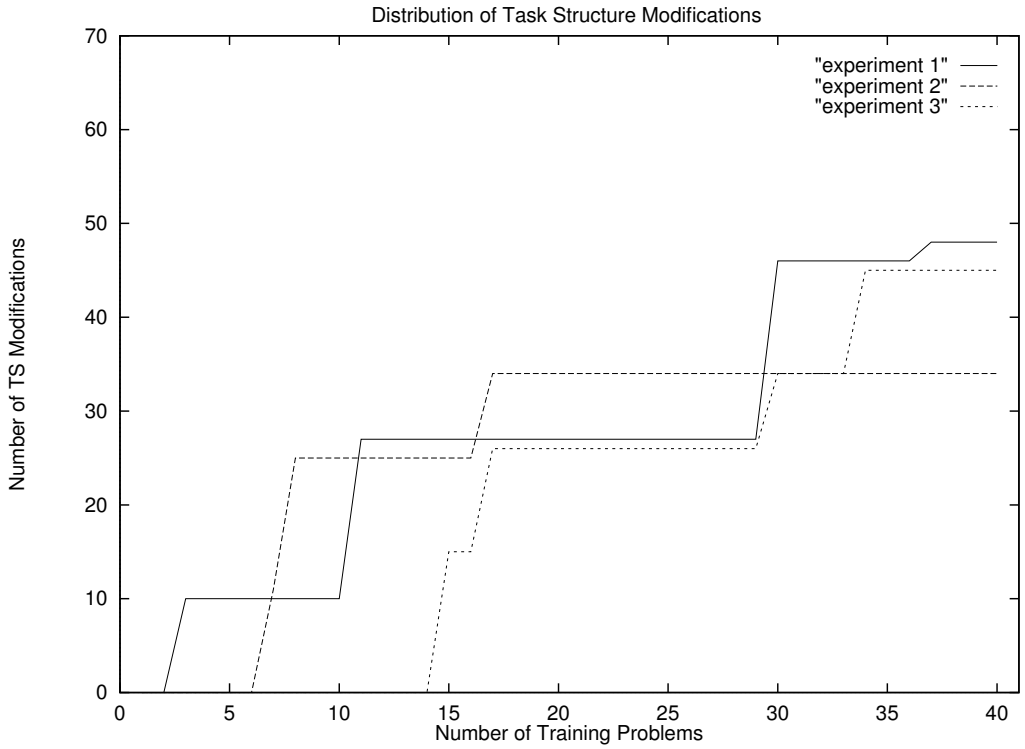


Figure 9.1: Distribution of Task-Structure Modifications in the "Quality of Solutions" experiment.

the path-increase , int-selection and path-retrieval task, where several possible alternatives may have been postulated. The short steps correspond to problems that caused the subsequent redesign of the methods by selecting one of these other already existing alternatives to substitute the originally chosen modification. The fact that in all six experiments AUTOGNOSTIC modified the same elements of ROUTER's functional architecture, and in consistent ways, indicates that given a particular design of a problem solver there are some elements that are more flexible than others, and these are the elements that are bound to get modified when new requirements are imposed on the problem-solver's performance. Thus AUTOGNOSTIC does not keep on modifying the problem-solver's task structure indefinitely. In fact, it stops modifying it when it has found (possibly after some exploration and redesign) a good design solution for modifying these flexible design elements. However, because the "right" knowledge context needs to exist in order for AUTOGNOSTIC to discover the appropriate redesigns, convergence may occur sooner or later depending on the problem-solving episodes that AUTOGNOSTIC reflects upon. Thus in the second quality-of-solution experiment, stability is reached at the 17th problem, where in the first and third quality-of-solution experiments it occurs much later. It is also interesting to note here that convergence occurs faster in the process-efficiency experiments than their quality-of-solution counterparts, although they face the same problem sequences. An explanation for that phenomenon is that different "optimization" criteria are more or less "natural" to the current problem-solver's design, and thus the redesign process towards "optimizing" them converges faster or slower correspondingly. This is true for these experiments with ROUTER since at no point during the design of ROUTER was optimality of paths (in any dimension) considered.

Figures 9.3 and 9.4 show that, as the problem solver solves more problems and gets more feedback on its performance, acquisition of new knowledge slows down, since the problem solver incrementally fills in its knowledge gaps.

On the other hand, knowledge reorganization modifications do not seem to stabilize in Figures

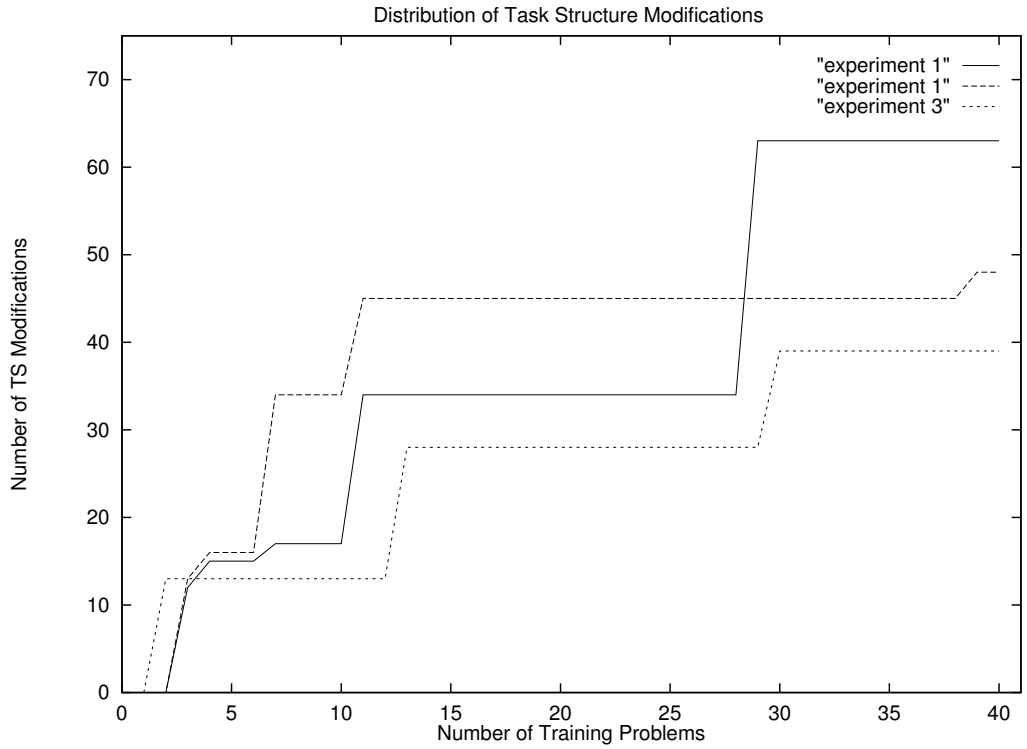


Figure 9.2: Distribution of Task-Structure Modifications in the "Process Efficiency" experiment.

9.5 and 9.6, representing the distribution of such modifications while training. One explanation of this non-convergence phenomenon is that, because the problem decomposition as performed by the int-selection task is commonly unsuccessful, AUTOGNOSTIC tries to expand the ROUTER's neighborhoods so much that all problems can be solved by intrazonal-search which is much more often successful in producing the desired path. This explanation suggests that the non-convergence of the knowledge-reorganization modifications is the result of an inherent conflict between the optimization criterion (which can be accomplished in "rich" knowledge context) and the problem decomposition (which imposes "artificial" knowledge localities).

9.5 Experience with AUTOGNOSTICONKRITIK2

The effectiveness of AUTOGNOSTIC's learning in the context of its integration with KRITIK2 was evaluated only with single problem-solving-and-learning episodes. KRITIK2's domain is much more complex than ROUTER's, and as a result, experimentation with long sequences of problems would require massive infusion of knowledge to KRITIK2, which is beyond the scope of this thesis. Table 9.7 summarizes these episodes and the performance improvement they result in. AUTOGNOSTIC's SBF model of KRITIK2's design, as used in the following examples, is shown in Appendix 2.

Example 8: Reorganization of the Task Structure: [4]

AUTOGNOSTICONKRITIK2 is presented with the problem of designing a sulfuric-acid cooler that will cool acid from temperature $t1$ to temperature $t2$, where $t2 < t1$. It retrieves from its memory the design of an existing sulfuric-acid cooler which cools acid from temperature $t1$ to $t2_{less}$ where

[4] This example is discussed as example 3 in greater detail in sections 5.2.3, 6.4, and 7.5.1

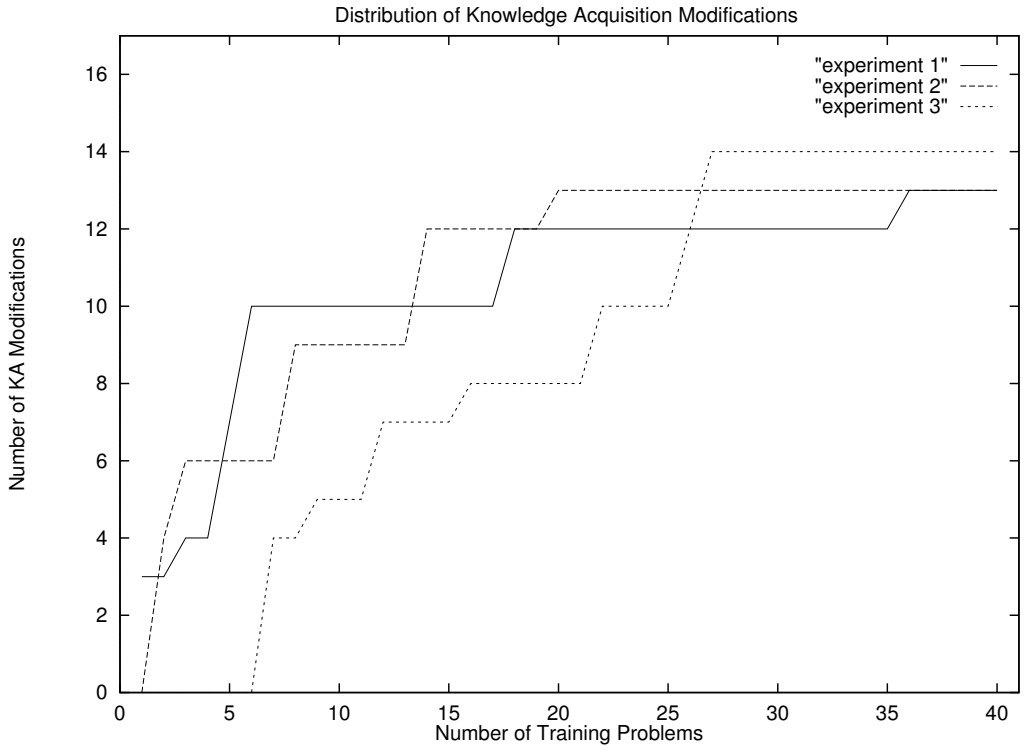


Figure 9.3: Distribution of Knowledge-Acquisition Modifications in the "Quality of Solutions" experiment.

Table 9.7: Learning from individual problem-solving episodes in AUTOGNOSTICONKRITIK2.

Example Problem	Performance Improvement
8	Problem-Solving Efficiency
9	Population of Solvable Problems
10	Quality of Solution, Population of Solvable Problems
11	Population of Solvable Problems

$t2_{less} < t2$. After diagnosing the existing design, AUTOGNOSTICONKRITIK2 decides that a potential cause for its failure to cool the acid to the desired temperature $t2$ is the capacity of pump which pumps the acid into the heat-exchange chamber. At this point, AUTOGNOSTICONKRITIK2 knows two repair plans which it can use to redesign the existing design so that it delivers the new function: it can either replace the failing component with a new one of higher capacity, which will pump the acid inside the chamber at a higher rate, or it can replicate the existing pump with the same affect to the flow rate of the acid. Both these plans are evaluated to be applicable in the current situation, and

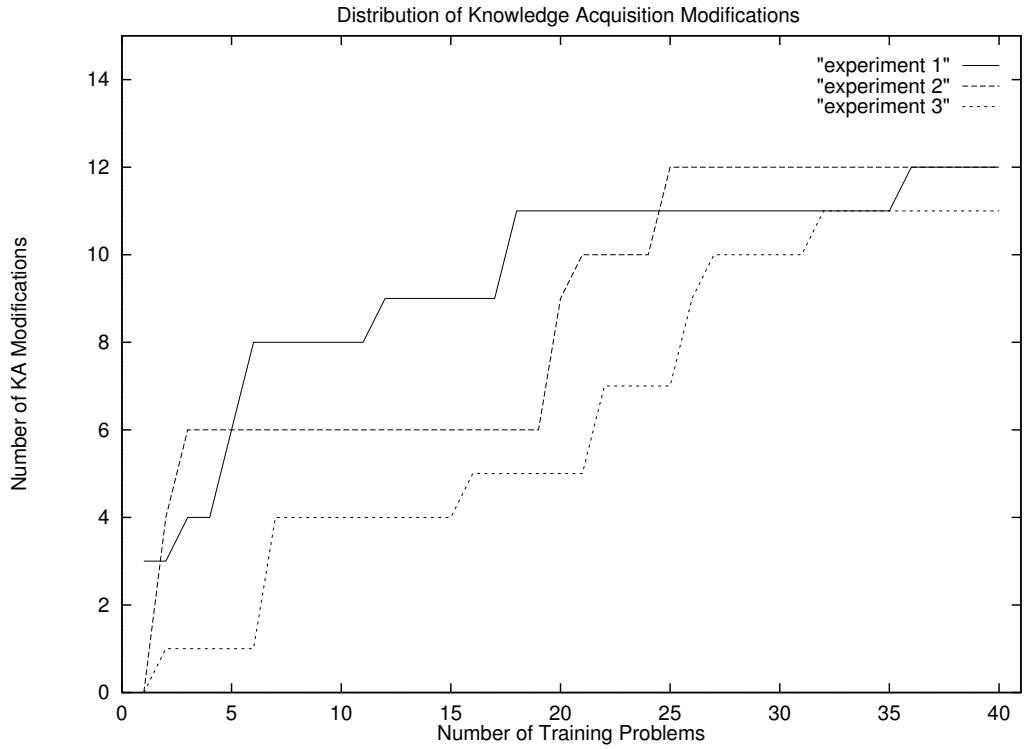


Figure 9.4: Distribution of Knowledge-Acquisition Modifications in the "Process Efficiency" experiment.

AUTOGNOSTICONKRITIK2 chooses the former one at random. In trying to apply the former plan, AUTOGNOSTICONKRITIK2 realizes that it does not have an appropriate component in its component memory which can be used as a replacement for the failing one, and fails to successfully repair the existing design.

At this point, AUTOGNOSTICONKRITIK2 recognizes that its understanding of the interdependencies among its tasks is not perfect. More specifically, it recognizes that the success of the component-replacement plan depends on the availability of an appropriate replacement component in its memory. AUTOGNOSTICONKRITIK2 makes that condition explicit in its task structure, and in order to do that it also reorganizes it, to bring the subtask t-get-component-fault-to-replace outside the plan.

When AUTOGNOSTICONKRITIK2 repeats its problem-solving for the same problem, it evaluates only the structure-replication plan as applicable to the redesign problem at hand, and proceeds to apply it successfully.

Example 9: Knowledge Acquisition: At the end of this second problem-solving process, AUTOGNOSTICONKRITIK2 is given as feedback the information that there exists a component which could have been used as replacement, i.e., the *big-acid-pump*. As it assimilates the feedback information, AUTOGNOSTICONKRITIK2 recognizes that the pump presented to it as feedback does not belong in its component-memory and thus it proceeds to update its domain knowledge by acquiring knowledge regarding the feedback pump. After, having integrated in its component-memory the new pump, AUTOGNOSTICONKRITIK2 repeats its problem solving and is able to use the component-replacement plan to repair existing design.

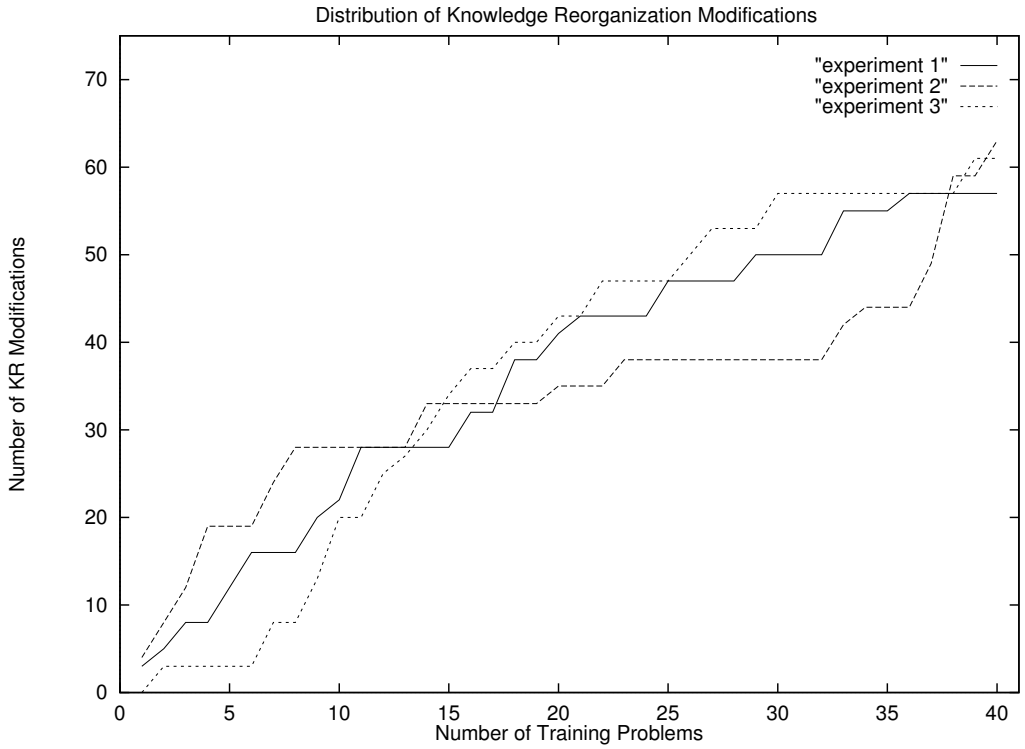


Figure 9.5: Distribution of Knowledge-Reorganization Modifications in the "Quality of Solutions" experiment.

Example 10: Selecting the "Right" Replacement: After having successfully completed the repair of the existing design with the replacement of its pump, AUTOGNOSTICONKRITIK2 receives as feedback that it should have used as a `replacement` yet another component, i.e., the *big-sac-pump*. This new component is similar to the component that AUTOGNOSTICONKRITIK2 actually used as replacement, in that they both have greater capacity than the pump in the existing design, and therefore they both can be used as `replacements`. However, the new pump is specifically built to be used for pumping sulfuric acid, where the *big-acid-pump* is a generic pump which can be used with all kinds of acids.

The original implementation of KRITIK2's `component-replacement` plan does not take into account the types of substances that a particular component allows through it, when it decides which component to use as a replacement of a failing component in a design. Often however, the correct functioning of the device depends on the choice of an appropriate component given the properties of the substances that this component is expected to allow through it. Such dependencies are often made explicit in the SBF model and are revealed during the subsequent stages of the revision of the model. This example makes explicit this omission in KRITIK2's design. AUTOGNOSTICONKRITIK2's reflection on its problem solving results in the recognition that, according to the semantics of the `get-component-fault-to-replace-hi gher -para m` there exist two components that can be used as a replacement of the failing pump in the existing design; however, one of these two components is preferable to the others, and therefore AUTOGNOSTICONKRITIK2's goal becomes to refine the functionality of this task, in order to produce the components of the preferred kind. AUTOGNOSTICONKRITIK2 recognizes that the difference between the two components in the current episode is that the preferred one is built "specifically" for the substance that flows in the design under repair, where the other one is generic. Therefore it decides to introduce a selection task in its `repair` task structure which will specifically choose the right component among all the

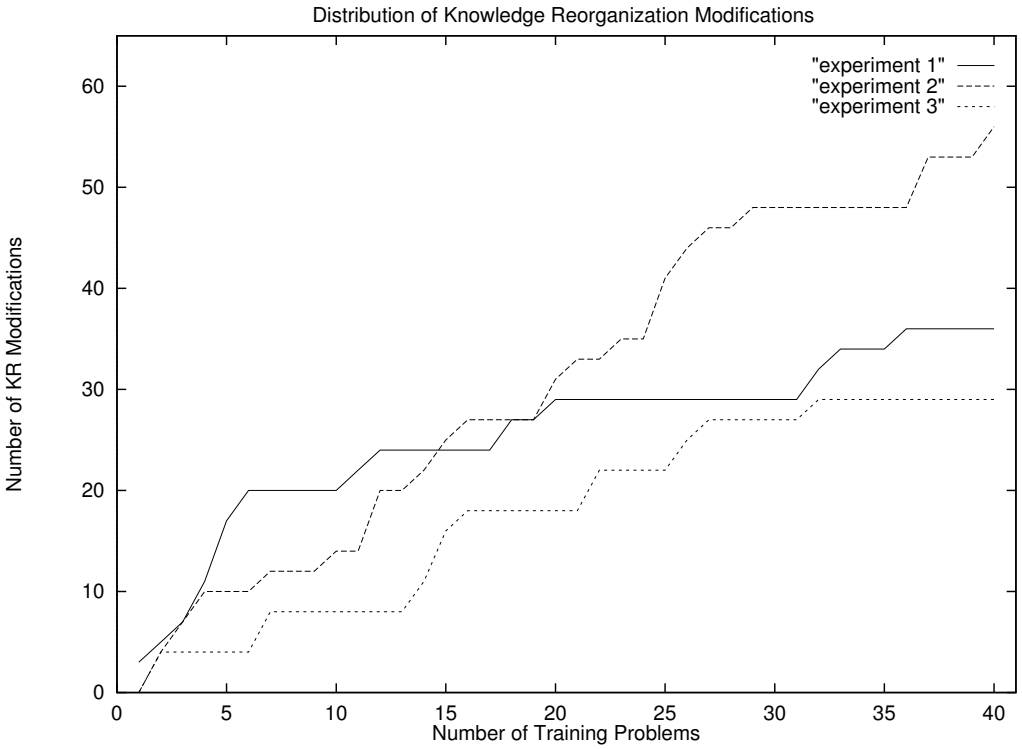


Figure 9.6: Distribution of Knowledge-Reorganization Modifications in the "Process Efficiency" experiment.

possible ones, i.e., the one which is built specifically to allow the flow of the substance which in fact will be flowing through it in the repaired device.

Example 11: Reorganizing the Design Memory: The designs in KRITIK2's memory are hierarchically organized along the types of substances that each particular design transforms through its functioning. Often, the functioning of a particular design involves the transformation of more than one substances. In such cases this design should be indexed under the memory nodes corresponding to all of the relevant substances, so that it can be retrieved when a problem about any of these substances is presented to KRITIK2. This however is not always the case. When KRITIK2 creates a new design and stores it in its memory, it indexes it only under the memory nodes that correspond to substances mentioned in its functional specification. Thus, sometimes a design may not be retrieved even though when it is relevant to a given problem, because its similarity with the problem at hand lies in a substance not mentioned in its functional specification.

AUTOGNOSTICONKRITIK2 faces this problem of incomplete memory organization when presented with problem of designing a water heater that will heat water from temperature $t3$ to temperature $t4$ where $t4 > t3$. AUTOGNOSTICONKRITIK2 cannot solve this problem because it does not have any devices in its design memory indexed by the memory node corresponding to designs that involve water. However, it does have in its memory a nitric acid cooler which heats water from temperature $t3$ to temperature $t4_{less}$ where $t4 > t4_{less} > t3$. AUTOGNOSTICONKRITIK2 could redesign this nitric acid cooler to heat water up to temperature $t4$ if it could retrieve it, but because of its design-memory organization it is not able to recognize the similarity of the nitric-acid cooler with the current problem. AUTOGNOSTICONKRITIK2 is given as feedback the information that a design that could be used as the basis for solving the problem at hand, old-case , could be the nitric-acid cooler. At this point, AUTOGNOSTICONKRITIK2 notices that it already knows about

this device, and the reason it did not retrieve it is because it does not meet the semantic relations of the get-cases-of-node task, which retrieves the cases associated with these memory nodes that have been found to be relevant to the problem at hand. Thus it suggests as a modification that would address the failure the reorganization of its design memory so that the cooler is indexed by the "water devices" memory node also. Indeed, the second time it attempts to solve the problem, AUTOGNOSTICONKRITIK2 successfully retrieves and redesign the nitric acid cooler.

Comments on the AUTOGNOSTICONKRITIK2 Experience
Examples 8, 9 and 10 seen together make explicit some of the interactions between the symptoms of the failure, the state of the problem solver, the feedback from the external environment, and the modification that AUTOGNOSTIC suggests to the failing problem solver. In all three examples, the problem is that KRITIK2 fails to use "the right" component as a replacement for the failing one in the existing device. Because, however, the state of KRITIK2's domain knowledge and the quality of the feedback that the external environment provides to AUTOGNOSTIC is different, the modifications performed to KRITIK2 are also quite different. In example 8, AUTOGNOSTIC is not given any feedback, and thus the only modification it can suggest is to reorganize KRITIK2's task structure, so that the inter-dependencies between the get-component-to-replace and the replace-component tasks are explicit. In example 9, the knowledge state of the problem solver is the same as in the first one, however, this time KRITIK2 is not mistaken and recognizes that the component-replacement plan is not applicable. In this example, the feedback from the external environment is more informative, i.e., it points to the actual component that could have been used as a replacement. Thus AUTOGNOSTIC is able to recognize the incompleteness of its domain knowledge, and can proceed to update it. With the updated domain knowledge, KRITIK2 has the option to use the component-replacement plan. Finally, in example 10, KRITIK2 also has the option of using that plan, although it can instantiate it in two different ways. Because its task structure does not include a task which explicitly reasons about the options and makes the choice between the possible instantiations, KRITIK2 selects the first possible choice. Given feedback from the environment which suggests that another possible choice should have been preferred, AUTOGNOSTIC guesses a criterion that differentiates between the possible choices and introduces a task which, based on this criterion, will explicitly decide on the "best" way to instantiate the plan.

Although, these four examples cannot be used as the basis for obtaining statistical results on the improvement of KRITIK2's performance due to its redesign by AUTOGNOSTIC, they, nonetheless, provide some evidence for the effectiveness and the generality of AUTOGNOSTIC's reflection. First, the modifications performed by AUTOGNOSTIC to KRITIK2 cover the range of modifications that AUTOGNOSTIC is able to perform. Thus, these examples provide evidence to the fact that AUTOGNOSTIC's modifications are not tailored to any particular task, domain, or problem solver. Furthermore, since each modification improved KRITIK2 to some extent (i.e., example 1 improved its efficiency, examples 2 3 and 4 the class of problems it can solve, and example 3 the quality of the solutions it can produce), these examples provide evidence for the generality of scenarios where AUTOGNOSTIC's learning tasks are useful.

9.6 Experience with AUTOGNOSTICINAURA

The third problem solver integrated with AUTOGNOSTIC is REFLECS. The rearrange-the-office task discussed in section 4.2 was one of the tasks in the AAAI-93 robotic competition. This task gave rise to an interesting requirement on the robot's behavior, that is, for some part of its behavior it had *to not avoid* a particular obstacle, i.e., the box that it had to move. For almost all other tasks that the robot ever performed, the avoid-obstacle behavior was always active, making sure that the robot did not bump into any solid objects in its environment. In this case, however, it was part of the task specification that the robot had to bump into a particular solid object. Thus, this task of the AAAI-93 competition naturally gave rise to a problem where the current configuration of the robot behaviors needed to be redesigned in a novel way to meet its requirements.

Two different design options were considered by the Georgia Tech robotics team for the AAAI-93 competition: first, to re-implement the avoid-obstacle behavior so as to make it sensitive to those objects in the robot's environment which were explicitly labeled as obstacles, and not to all of them; second, to completely switch off this behavior for as long as the robot had to hold the box. The second option was chosen for the competition and the overall rearrange-the-office task was accomplished by the process described in section 4.2. Later, the selective avoid-obstacle schema was also implemented, so it was available for the REFLECS experiment.

For this experiment, we presented[5] REFLECS, the integration of AUTOGNOSTIC with the reactive planner described in Chapter 4.2, with the task of getting to a particular box. REFLECS was able to get close to the box and then it started oscillating in small steps around it. This was due to its active avoid-obstacle behavior which repulsed it from the target. At this point, AUTOGNOSTIC recognized that no progress was being made in the execution of the task, since the motion of vector synthesized by the move-to-goal and the avoid-obstacle vectors was insignificant, and proceeded to interrupt the planner's behavior and assign blame for this failure.

AUTOGNOSTIC noticed that the reason no significant vector was produced was because the task responsible for producing it, synthesize vectors, does not get accomplished when the vectors it takes as input are of opposite direction and same length. Thus, it inferred as a potential cause for the failure the over-constrained functionality of the synthesis task and suggested as a possible modification its replacement with another synthesis task which would produce a vector even when its inputs are same-length, opposite vectors. This modification was not possible because there is no other instantiation of the synthesis task available at the reactive level. It is interesting to note, however, that this suggestion could be implemented as a new synthesis task which under these conditions would produce a "noise" vector. This solution would address the problem of non-progress, but it would not fix it in the long run. As an alternative, AUTOGNOSTIC proposed the falsification of the condition which causes the synthesize vectors task not to occur. One way the condition can be falsified is if the information on which the which applies, i.e., the move-to-goal-vector and the avoid-obstacle-vector, has different values. This, in turn, could be accomplished, if the tasks that produced these types of information were different. Thus, AUTOGNOSTIC proposed the instantiation of alternatives for the move-to-goal and the avoid-obstacle schemas. This modification was possible since there is an alternative instantiation of the avoid-obstacle schema which is more selective and does not produce a repulsive vector for the goal object. Indeed, AUTOGNOSTIC suggested the replacement of the currently active avoid-obstacle with this more selective one, avoid-obstacle-selective, and subsequently allowed the planner to proceed with its task. Indeed, after this replacement, the planner reached the box.

Comments on the AUTOGNOSTICINAURA Experience

This example evaluates AUTOGNOSTIC's reflection process with yet another agent, quite different from the class of problem solvers which were originally thought to benefit from it. Furthermore, it gives some insights to some issues in the periphery of this thesis, but not directly addressed by it. For example, what is the interaction between the reflection process and the process responsible for the reasoning behavior of the agent? In AUTOGNOSTIC's integration with ROUTER and KRITIK2 the reflective monitoring process and the reasoning process were completely integrated. In AUTOGNOSTIC's integration with the AuRA reactive planner, the planning behavior occurs independently from the reflective monitoring, although the two are synchronized, and only when there is a problem the reflection process interrupts the planning process and takes over. Another issue partially dealt with in this example was the question of whether the task-structure modifications are permanent or not. In AUTOGNOSTIC's integration with ROUTER and KRITIK2 these modifications were permanent. In AUTOGNOSTIC's integration with the AuRA reactive planner, both the original and the modified task structures are retained, the latter one annotated by the symptoms of the failure and the modification that produced it. The original task structure becomes the default option when the agent is asked to accomplish a specific instance of the task at hand. If however there is a failure, similar to the one addressed previously, the agent may automatically switch to the modified task

[5] This experiment was conducted with a simulator of the robot.

structure without going through the blame-assignment and repair processes.

9.7 Realism

Different theories of learning propose different types of knowledge as necessary for the learning behavior to occur. For example, rules, cases and models are just few examples of the different types of knowledge that have been suggested as necessary for learning. In addition, these theories assume the availability of these types of knowledge to a varying degree. Many theories assume the existence of the necessary knowledge at "boot-strapping" time, while others include mechanisms for incrementally acquiring some of the knowledge they need. Finally, if for some reason the particular knowledge they require becomes unavailable, different theories degrade in different dimensions. Thus, a very important dimension, in which it is interesting to analyze a learning theory, is in terms of how realistic its knowledge assumptions are. More specifically this issue of realism has three aspects:

1. How plausible it is that the knowledge the learning process requires is indeed available?
2. How realistic is the type of feedback it assumes?
3. To what degree can the theory explain the acquisition of the knowledge it requires?
4. In what aspects does the theory degrade when this knowledge is missing?

AUTOGNOSTIC's reflective learning process assumes the existence of the SBF model of the problem solver that is being reflected upon.

The first issue then becomes, the plausibility of AUTOGNOSTIC's assumption regarding the availability of SBF models of problem solving. From a cognitive-science point of view, there seems to be much evidence for meta-cognitive knowledge, reflection, and its beneficial affects to learning. The psychological research on reflection is discussed in greater length in chapter 10. From an artificial-intelligence point of view, recently, a series of tools have been developed that support the development and implementation (but not the redesign) of intelligent systems through analysis similar in nature to task-structure analysis. Such tools include KREST [Steels 1990, Tadjer 1993, van Nijnatten 1993] and Spark-Burn-Fire-fighter [Marcques *et. al* 1992]. The use of such tools to build AI systems implies the existence of SBF-like models for the resulting systems.

Regarding the realism of the type of feedback required by AUTOGNOSTIC, the preferred solution to a given problem may be costly to produce than a simple success or failure type of feedback, but it is also more informative. On the other hand, it is less costly than the complete trace of executing the actual solution or the complete trace of producing the preferred one. Furthermore, it enables the reflective system to benefit from feedback originating from experts which do not use the same reasoning strategies with the system, which is impossible in the case of feedback which includes a complete trace.

Regarding the issue of acquisition of these SBF models of problem solving, AUTOGNOSTIC provides an initial account of how these models can be incrementally learned from existing incorrect SBF models. If the SBF model is incorrect, i.e., the expected behavior of a task is incorrect, AUTOGNOSTIC can recognize the inconsistencies between the actual successful behavior and its description and it can postulate new descriptions. AUTOGNOSTIC does not have an account of how it can acquire such models for existing imprecise ones, or from scratch.

Finally, the third question, with respect to the knowledge assumptions of AUTOGNOSTIC's reflection, that must be addressed is how the quality of the SBF model of the problem solver affects the effectiveness of the reflection process, and more specifically how poor quality SBF models cause degradation of its effectiveness.

There are three important aspects to the quality of a SBF model:

1. the quality of the task-structure analysis of the problem-solving process, i.e., the level of detail and the level of precision in which the problem-solver's reasoning process is described,

2. the quality of the meta-model of the problem-solver's domain knowledge, i.e., the level of detail in which the domain knowledge that the problem solver has available to it is described, and

3. the degree to which the level of these descriptions match, i.e., the degree to what the semantic relations of the problem-solver's subtasks can be expressed in terms of domain relations it knows about.

If the SBF model is not precise, i.e., if the expected correct behavior of the problem-solver's tasks is poorly described, (i.e., sparse functional semantics or not at all), then the blame-assignment method has no basis of evaluating whether or not the feedback solution is within the class of solutions the task structure is able to produce. Thus, it does not have any basis for suggesting modifications to the task structure. Furthermore, it does not have evidence to suggest changes to the domain knowledge either, since changes to some part of the domain knowledge are suggested after there is evidence for the failure of some task that uses this domain knowledge.

If the SBF model does not explain in enough detail the problem-solver's subtasks, i.e., if it only analyses its process in terms of big, complex subtasks, then the blame-assignment method may not able to localize enough the potential cause of the failure. If the grain size of the described tasks is too big, then they are bound to be complex and thus they are bound to play multiple roles in the context of the overall problem-solving task. Thus, even if the fault is localized within such a big complex subtask, the blame-assignment method may not be able to suggest operational modifications. Even if it does suggest modifications they are bound to have undetectable consequences and thus lead to inconsistent problem-solving task structure.

If the meta-model of the problem-solver's domain knowledge is described in little detail, then the blame-assignment method may not be able to trace failures of subtasks to errors in the domain knowledge. This is because the delegation of the responsibility for a failure, from a task to some part of the domain knowledge, relies upon the description of the task semantics in terms of specific domain relations. The fewer relations the SBF model describes, the fewer inter-dependencies it will be able to express between the task structure and the domain knowledge. Consequently, the reflective system may not be able to improve its domain knowledge. Also if the meta-model of the problem-solver's domain knowledge is poor, then the reflective system may not be able to recognize new uses for this domain knowledge and consequently it may not be able to introduce new tasks in the task structure or modify the semantics of the existing tasks. Finally, if the domain meta-model does not make explicit constraints among types of domain knowledge, then some modifications to the domain knowledge are bound to lead to inconsistencies with other existing pieces of knowledge. Finally, the detail with which the meta-model of the problem solver is described decides the space of possible functional concepts that AUTOGNOSTIC can postulate while modifying the problem-solver's task structure. Thus, a poor meta model may also limit the range AUTOGNOSTIC's ability to infer new tasks to integrate in the problem-solver's task structure.

From this discussion it must also be evident how important a good match between the task structure and the domain meta-model is critical to the effectiveness of the reflection process. If the meta-model of the problem-solver's domain knowledge is described in terms different than the semantics its tasks, then the blame-assignment method may not be able to trace failures of subtasks to errors in the domain knowledge or to integrate new uses of its existing domain knowledge in the task structure.

9.8 Limitations

The extensive experimentation with AUTOGNOSTIC also revealed some limitations.

9.8.1 Limitations of the System

Expressiveness of Task Semantics: To date AUTOGNOSTIC's language for describing task semantics (shown in Figure 7.16) can express only unary and binary relations between two different

types of information. This was an implementation decision made while describing ROUTER's SBF model, because at the time it seemed sufficient, at least for ROUTER. This decision limits the class of inferences that can be described in the SBF modeling framework and consequently the class of problem solvers that can be modeled.

In fact, after careful examination of the reasoning behavior of the original ROUTER and AUTOGNOSTICONROUTER, it turned out that this decision limits even the faithful description of ROUTER. Let us see how. In the original implementation of ROUTER, all the knowledge captured about its world model was implemented in a single data structure and several aspects of this knowledge were implicit in the implementation of this data structure. In ROUTER a street can be mentioned in several different neighborhoods, and its description in each one of these neighborhoods is different because each neighborhood covers a different space at different levels of abstraction. The information about which intersection of a certain street is next to which other intersections in some particular neighborhood is captured implicitly in the ordering of the intersections in the description of a street in some particular neighborhood. Notice, that this relation is ternary: it maps each intersection mentioned in some neighborhood to the intersections next to it in that neighborhood. Notice also that when ROUTER expands its current path (in each cycle of the increase path task) it generates new possible paths paths which relate to its current path with the following relation: *ForAll np ∈ new-possible-paths: final-node(np) ∈ next(final-node(cp), z)* where *z* is the neighborhood in which the search is performed. The current grammar for task semantics does not allow this expression.

If the SBF language were extended to allow the specification of higher-order domain relations, two modifications would be necessary to extend AUTOGNOSTIC's language for task semantics:

1. extension of the process of inferring alternative task inputs when its actual inputs and desired output do no meet its semantics, and

2. extension of the relation-discovery process to postulate higher order relations among the types of information available in the problem-solving context.

The second modification is straightforward. AUTOGNOSTIC generates possible relations by filling in the templates of the semantic relations it can express with information available in its problem-solving context. Therefore, if its language included additional templates for higher-order relations, it would simply have to identify higher-order combinations of information types that could be used to fill them in.

The first modification is slightly more interesting. AUTOGNOSTIC currently infers alternatives for its inputs by looking at the truth table of the domain relation to which the semantic relation refers, or, if the relation is evaluated by a predicate, by applying the inverse predicate of this relation, or by searching the domain of the input for a value that would verify the failing semantic relation. If a failing semantic relation related some particular output to several inputs, and this relation referred to a domain relation with a truth table, then AUTOGNOSTIC would have to search this table for these tuples which included the desired output. For example, let us assume the semantic relation *o ∈ f(i j)* and a particular failed problem-solving episode, where the desired output *o2* and the actual inputs *i1* and *j1* conflict with this relation. If the relation *f* is evaluated by a truth table, then AUTOGNOSTIC would have to identify the entries of the form *?i ?j o2* in the table. The search might be guided by the actual values for the input, for example, AUTOGNOSTIC might search first for the tuples *?i j1 o2*, then for the tuples *i1 ?j o2*, and finally for the tuples *?i ?j o2*. If the relation is evaluated by a predicate, if there is and inverse predicate, the process would be exactly the same as it is now. If not, there might be predicates for inferring the value of one input given the output and the rest of the inputs which AUTOGNOSTIC might use. For example, if relation *f* was evaluated by a predicate, and there existed predicates *g: j ∈ g(i o)* and *h: i ∈ h(j o)*, then AUTOGNOSTIC could use these predicates to infer alternatives for the values of *i* and *j*. Finally, if none of these cases are applicable, the search in the domain of the inputs would still be an option, only the search would have to be conducted in the Cartesian product of the input domains. Therefore, extending AUTOGNOSTIC's language for expressing task semantics is possible although at the cost of additional complexity.

9.8.2 Limitations of the Theory

Redesign of the problem-solver's high-level tasks: AUTOGNOSTIC modifies the internals of the problem-solver's task structure, but not the overall task of the problem solver. Changing the input of the overall task is within AUTOGNOSTIC's capabilities. Let us assume that more information is given to the system as part of its problem specification than what the task requires as input. Let us also assume that this information is "interpretable" given the current domain ontology of the system. To the extent this information is useful in adapting the system's performance, Autognostic is able to "discover" its appropriate uses and integrate it in the task structure.

Changing the overall-task output is more complicated. In general, to produce completely different outputs than what it is designed for, the system will need to develop different internal processing mechanisms. Autognostic is able to deal with incremental modifications to the system's existing mechanism, and not with building processing mechanisms from scratch. One possible approach to that problem, could be by analogy to other "similar" processing mechanisms. Such an approach would be mediated by the concept of "prototype tasks" that already exists in the SBF modeling framework. More specifically, if AUTOGNOSTIC knows about tasks that are instances of the same prototype but have different number of inputs and outputs, when new outputs are required of one of these instances it could attempt to transfer by analogy mechanisms from other instances that produce similar outputs.

Redesign of ontology: To date, AUTOGNOSTIC has only the ability to recognize the potential failure of its domain ontology to express all the instances of objects that actually exist in its domain, but it cannot redesign its ontology. The reason is that AUTOGNOSTIC does not currently have a theory of how primitive concepts may be composed into more complex ones. Because it has a language (quite limited in the current implementation) for expressing the current ontological commitments it is able to recognize their failure. However, in order to postulate modifications when they fail, and to effectively redesign them, it would have to have a compositional theory. This problem of re-representation of domain knowledge is a very difficult open research issue in itself.

9.9 Summary

The table of Figure 9.8 summarizes the evaluation of AUTOGNOSTIC's theory of reflective learning.

Table 9.8: Summary of AUTOGNOSTIC's Evaluation.

Issue	Experiment	Comments
Is it general?	3 different systems, i.e., 2 deliberative (Router, and Kritik2) 1 reactive (Reflecs)	Identifiable design elements Specifiable Functions Specifiable Composition
Is it effective? • Set of Solvable Problems	ROUTER QoS experiments Efficiency experiments KRITIK2 Examples 9, 11 REFLECS experiment	KA enables PS in a larger domain Also, KR improves knowledge access
• Quality of Solutions	ROUTER QoS experiments Examples 1, 4, 5 KRITIK2 Example 10	Quality criterion expressible in terms of the agent's knowledge
• Problem-Solving Efficiency	ROUTER Efficiency experiments Examples 6, 7 KRITIK2 Example 8	May require more localized feedback
Does it converge?	ROUTER QoS experiments Efficiency experiments	Consistent feedback
Realism • Incorrect SBF model? • Incomplete SBF model?	It can recognize the error and potentially correct it Problems	ROUTER example 2 See section 9.7

CHAPTER X

RELATED RESEARCH

The work of this thesis is related to several lines of research in machine learning, artificial intelligence, and cognitive science. This chapter delineates its relations to earlier AI research on modeling problem solving, learning from problem-solving experiences, reflection and learning, and psychological research on reflection. In addition, because some of the ideas that gave rise to the original research hypotheses have been conceived in the context of design of physical devices, some analogies are drawn between this work and design and modeling of physical devices.

10.1 Modeling Problem Solvers

The need for several levels of description of the knowledge contained and used in intelligent systems was identified quite early. Newell [Newell 1982] suggested that information processing should be characterized *at the knowledge level*; at that level, knowledge is described functionally, in terms of what it does, and not structurally, in terms of the particular objects it refers to and their relations. Marr [Marr 1982] too described four levels of analysis of intelligent systems, in order to separate implementation decisions from the computational theory that these systems embody. He emphasized the need for analysis of expertise in terms of the tasks accomplished, how they are organized, and what is the knowledge required by each task.

10.1.1 Generic Tasks

AI research in knowledge systems, in particular, has led to several theories that emphasize the concept of "task" in analyzing problem solving. For example, Chandrasekaran [1983] suggested that tasks fall into major classes, called generic tasks. In specific fields of expertise, tasks are instances of these generic tasks. All the tasks that fall under the same generic task share the same decomposition into simpler subtasks, that is, they are accomplished by the same basic reasoning steps, and require similar types of domain knowledge.

In a similar vein, Clancey [1985] analyzed the inference structure underlying a series of expert systems, showing how the heuristic classification framework applied to each one of them. He also showed [Clancey 1986] that making the underlying inference structure explicit in terms of tasks and meta-rules, in a way similar to generic tasks, enables transfer across domains and simplifies the knowledge-acquisition process.

McDermott [1988] started developing a series of knowledge-acquisition tools, with emphasis on the problem-solving method. He proposed that a *role-limiting method* can be used to perform families of tasks by abstracting the control knowledge from their peculiarities. Such a method defines highly regular roles for the knowledge necessary to perform a particular task, and thus provides strong guidance as to what knowledge is required and how it should be encoded. Finally, McDermott pointed out that the clarity of the knowledge roles, and thus the strength of the method's guidance, diminishes with the increasing diversity of the information required to identify candidate actions and choose among them. Role-limiting methods are similar to generic tasks, in that they both postulate generic classes of reasoning tasks, and to support this hypothesis, they both point to patterns of regularity in the reasoning of experts from different domains on different tasks, and to common types of knowledge that these experts use while performing their tasks.

Finally, Wielinga and his colleagues [Wielinga and Breuker 1986, Wielinga *et al.* 1992] describe a four-layer model of expertise, KADS. In KADS there are different generic types of knowledge that play different roles in the reasoning process and they can be organized into several layers with limited interaction between them. The types of knowledge they describe are domain knowledge, and control knowledge of three types: knowledge of different types of inferences that can be made in the domain, elementary tasks and strategic knowledge. Several different tasks have been modeled in terms of the KADS framework and these models have been compiled in a book, which has been used by system developers as a guidance in the design of new systems.

All these approaches were developed in an effort to support transfer of the same basic pattern of inferences across different domains of problem-solving behavior. Thus, they emphasize mainly the uniformity across reasoning tasks, and they generally ignore the possibility that there can be multiple types of reasoning strategies which an expert can employ in the service of a single class of problems.

10.1.2 Task Structures

To enable the modeling of more flexible reasoning processes, the nuances of which cannot be part of a "generic task" model, Chandrasekaran [1989] described task structures as a way of building intelligent systems and modeling cognition. The structure of a task is a recursive decomposition in terms of the methods applicable to it, the subtasks they set up, the methods applicable to the subtasks, and so on. A task is specified by the information it takes as input and the information it gives as output. A method is specified by the type of knowledge it uses, the subtasks it sets up, and the control it exercises over the processing of these subtasks. The idea of task structures was a natural development of the generic-tasks idea, where generic tasks would constitute the building blocks of task structures. The analysis of a reasoning process in terms of its task structure makes explicit

1. the general methods that can be employed in accomplishing the overall goal of this process,
2. the types of domain knowledge that each step uses,
3. the role that each step of process plays in the context of accomplishing the overall goal, and
4. how this step depends on other steps of the process.

AUTOGNOSTIC uses the explicit representation of the task-structure of the reasoning process as a model of the actual reasoning process which explains the behavior of the problem solver while reasoning on each particular problem.

Steels [1990] proposed the componential framework for task analysis. This framework also proposes a detailed task analysis, in a vein more similar to task structures than generic tasks or role-limiting methods. Each task is analyzed from a conceptual perspective and a pragmatic one. From the conceptual perspective a task is described in terms of its input, its output and the nature of the operation it accomplishes, which corresponds roughly to a generic task. AUTOGNOSTIC also describes these three aspects of each task, with the difference that it admits the possibility that the nature of the operation of a task may not be possible to describe in terms of generic tasks, but may be described in terms of semantic relations, sometimes domain dependent. From the pragmatic viewpoint, the constraints imposed on the accomplishment of the task are described, such as time and space resources, quality of their domain knowledge etc. In this framework, tasks are accomplished by methods which employ two types of knowledge: case models, i.e., knowledge related to particular problem-solving situations, or more general domain models, i.e., domain knowledge independent of any particular situation. KREST [van Nijnatten 1993, Tadjer 1993] is a tool for modeling and developing systems, that integrates the componential-framework ideas and the KADS framework. In a similar evolutionary trend, McDermott and his colleagues developed Spark-Burn-Fire-fighter [Marcques *et. al* 1992], another tool for modeling the activity in a workplace, and developing systems to support it.

All of the above models of problem solving attempt to describe expert performance in domain-independent terms that make the control of processing more transparent. Such descriptions are very useful because they

1. facilitate transfer of inference mechanisms across domains,
2. make explicit the assumptions hidden in the implementation of systems that exhibit expert performance, and thus give more insight in their design and facilitate their maintenance, and
3. abstract away the domain-dependent details of the knowledge and identify the different roles that the knowledge plays in the performance of the task.

10.1.3 Using Task Models of Problem Solving

In addition to giving rise to tools for developing systems, these ideas for modeling problem solving have also been used to support other tasks, such as guide knowledge acquisition, enable explanations of expert-system performance [Chandrasekaran *et al.* 1989], assign blame for the failure of knowledge-based systems [Weintraub 1991], and modeling human expertise and monitoring the problem-solving behavior of students.

A very important precondition for extracting knowledge from human experts and using it in artificially intelligent systems is to have a vocabulary for expressing expertise which consists of terms that the experts can recognize and use, and which is precise enough that can be formalized computationally. Task-structure models specify expertise in terms of the methods that can be used to accomplish a task, the types of inferences that these methods require, and the types of domain knowledge on which these inferences rely. This vocabulary has been quite useful for humans to express the control and domain knowledge needed for accomplishing their tasks [Bylander and Chandrasekaran 1987, Chandrasekaran 1989].

A very important problem in building integrated systems where humans and machines cooperate to address a complex task is that, often, humans do not comprehend the systems' reasoning. As a result, they are often unable to interact effectively with them or to "trust" their performance. Specifying the systems' reasoning in terms of task models provides a description of their performance which is far more explanatory than a simple trace, and thus can better support this interaction.

solver for [1991] of

devices, Functional also very similar same agnostic of the software system.

Task models have also been shown quite effective in enabling the assignment of blame for failures in the reasoning of intelligent systems. Weintraub [1991] had used such models successfully to localize the causes of failure in an expert diagnostic system. This work is discussed in further detail later.

RedCoach [Johnson 1993] uses a task-structure model of a student's behavior to monitor its performance. The goal of RedCoach is to interpret the actions of the student in terms of a task structure, and thus recognize potential errors in its domain knowledge or in its strategic knowledge, and consequently suggest corrective courses of action. To date, RedCoach is able to recognize deviations of the student's problem solving from the "ideal" problem solving as described by its model. RedCoach can pursue multiple possible correct courses of action which can lead to the solution. Thus, it allows the student explore without stopping him/her prematurely, without wrong interventions or false alarms.

10.2 Learning and Problem Solving

AI research has investigated learning in two different contexts: as an autonomous process largely independent from any other activity of the agent [Martin 1992, Quinlan 1986], as well as in the context of an integrated system capable of problem solving and learning [Bhatta 1995, Carbonell 1986, Carbonell *et al.* 1989, Goel 1991b, Kolodner 1987, Laird *et al.* 1986, Mitchell *et al.* 1981, Mitchell *et al.* 1989, Redmond 1992, Samuel 1959]. In the

latter body of research, problem-solving needs define what is to be learned, and the learning process produces knowledge that can be used by the problem-solving process. Subsequent problem solving evaluates the quality and usefulness of the product of learning.

In general, systems with integrated problem-solving and learning capabilities can be compared along the following dimensions:

1. conditions under which learning occurs,
2. learning tasks, i.e., adaptations that learning induces on problem solving,
3. learning methods and types of knowledge that the learning mechanism uses, and
4. dimensions in which learning can improve the problem- solving performance.

Different AI systems have investigated different types of situations as conditions triggering learning, such as after a problem-solving episode, or after a single problem-solving step, or after a failure. Much work in AI has focused on failure-driven learning, not because learning does not happen in other situations, but because failure signifies a definite need for learning. A major step in the process of failure-driven learning is the assignment of credit (or blame) [Minsky 1963, Samuel 1959].

Learning mechanisms can be broadly distinguished in two categories, in terms of the modifications they perform to the problem-solving mechanism: they can introduce new knowledge in the system (knowledge-level learning) or they can transform knowledge from one form to another (symbol-level learning) [Dietterich 1986]. Learning at the knowledge-level can be, again, of two types: learning of *domain knowledge*, that is learning facts about entities that exist in the domain and their properties, or learning of *strategic knowledge*, that is new operators for action in the domain or new types of inferences for reasoning about it. A system may also be able to learn episodic knowledge in the process of problem solving: that is it can acquire knowledge about which objects, which inferences, and which actions were involved in a single reasoning episode.

Several different types of knowledge have been proposed as useful for supporting learning, and several different learning methods have been developed based on these types of knowledge. Such types of knowledge include problem-solving traces [Carbonell 1986, Mitchell *et al.* 1981], cases of particular problems solved by the system along with the their solutions and their outcomes when they were used afterwards [Hammond 1989, Kolodner 1993, Redmond 1992], causal knowledge of the domain of problem solving [Bhatta 1995, DeJong and Mooney 1986, Goel 1989, Mitchell *et al.* 1986], and different types of models of the problem-solving process itself [Davis 1977, Sussman 1975].

Most AI systems have focused on improving the efficiency of the problem-solving process, with a few of them also addressing the issue of solution quality. The issue of extending the class of problems a system can solve has been largely un-investigated.

To briefly summarize the position of this research in this four-dimensional space, the reflective learning process developed in this thesis constitutes a mechanism which improves the problem-solver's performance in terms of efficiency, quality of solutions, and population of solvable problems. This process is able to learn at the knowledge level. In addition, it is able to learn domain and strategic knowledge. Through the information it is given by an oracle, both in the form of problems and in the form of desired solutions, it recognizes the incompleteness/incorrectness of its domain knowledge and by assimilating this information it acquires domain knowledge. In its effort to explain why its problem-solving mechanism did not produce the desired information, it recognizes the incompleteness/incorrectness of its functional elements, and through its redesign process acquires new functional knowledge. This process relies on the comprehension of the problem-solving mechanism in terms of SBF model, and can utilize feedback from its environment in the form of the "correct" solution. Finally, this process is triggered by failure.

In this section, a set of problem-solving-and-learning systems are analyzed and their learning process is compared with AUTOGNOSTIC's in terms of the above-mentioned dimensions.

Lex The functional architecture of Lex [Mitchell *et al.* 1981] consists of four major elements: the problem solver, the critic, the generalizer, and the problem generator. The *problem solver* of Lex begins with a complete set of primitive design elements, i.e., operators, for solving symbolic-integration problems. However, the applicability conditions of these operators are not completely defined. Thus, in the beginning Lex performs symbolic integration through a blind forward search in this set of operators. Because this search is unguided, the problem solver of Lex pursues all possible operator application at each step in its reasoning. Thus, at the end of a problem-solving episode it has pursued a set of reasoning paths, some of them successful, some of them not. At this point, the problem solver passes the trace of its problem solving to the *critic*.

The critic identifies the best reasoning path of the paths pursued, and proceeds to identify and eliminate the causes for the inefficiencies of the other paths. The critic compares each one of the unsuccessful and suboptimal reasoning paths with the optimal path. Through direct comparison of these two traces, it identifies instances of positive and negative examples of correct operator applications. The successful trace provides the standard against which the unsuccessful ones are evaluated. All the operator applications leading in the optimal path are considered examples of correct operator selection, where all the ones leading out of this path and into dead-ends or suboptimal paths are considered to be examples of wrong operator selection.

The critic then passes these instances to the *generalizer*, which uses them as training examples on the basis of which it generates hypotheses for more informed operator-applicability heuristics. The learning method, i.e., candidate elimination, used by the generalizer to generate concepts defining the applicability conditions of a given operator allows the learned concepts to be more or less specific. When the applicability conditions of an operator are ill-defined, Lex's *problem generator* generates problems in order to refine the currently under-specified heuristics. To be able to design such pedagogical experiments, the problem generator has to inspect the current state of the problem solver, and to also take into account the internal workings of both the critic and the generalizer.

There are several interesting differences between Lex's and AUTOGNOSTIC's learning processes. The critic in Lex performs a kind of blame assignment on each unsuccessful reasoning path, based on the shortest successful path. For this task, the critic inspects each step of the failed trace and compares it with the corresponding step in the successful one. As Lex does not have an understanding of the role that each operator application is meant to play in the context of solving the overall problem, it can only evaluate its success or failure based on whether or not the same operator has been used at the same step in the successful trace. This blame-assignment method is fundamentally different from AUTOGNOSTIC's blame assignment. AUTOGNOSTIC's SBF model of the problem-solving process constitutes an abstract, characterization of a "correct" problem-solving process, independent of any particular problem. Thus, AUTOGNOSTIC does not require a complete successful reasoning trace in order to assign blame for the problem-solver's failure. Furthermore, since the SBF model is hierarchically organized, AUTOGNOSTIC is able to inspect the problem-solver's behavior at several levels of abstraction and detail, and it can acquit large segments of the problem-solving process without evaluating each individual step involved in these segments.

Another difference between these two theories is that, when Lex's generalizer hypothesizes new operator-selection heuristics it does not commit early to a single concept, but it rather keeps a space of possible concepts and refines it as needed. On the other hand, AUTOGNOSTIC commits early to one of the possible functional concepts it discovers and integrates it in the problem-solver's task structure, although it keeps its alternatives in case the chosen concept proves to be inappropriate. Where Lex incrementally refines the space of the candidate concepts, AUTOGNOSTIC selects which it may have to revise later.

The list below summarizes the comparison between Lex and AUTOGNOSTIC:

1. With respect to the kind of feedback that the two theories require, Lex requires the trace of the production of the preferred solution, where AUTOGNOSTIC requires the preferred solution itself. Lex is able to produce its own feedback by exhaustively exploring its problem space, where AUTOGNOSTIC is able to do so only sometimes. Lex's ability to produce all the knowledge it needs for learning stems from its assumption that the desired behavior is already within its competence.

2. The blame-assignment task in Lex cannot identify incorrectly specified operators, it can only

identify whether these operators were applied in the right situation or not. Therefore, Lex, unlike AUTOGNOSTIC, is unable to adapt the original set of operators (design elements) with which it is endowed.

3. With respect to the knowledge they require for the blame-assignment task, AUTOGNOSTIC requires, in addition to the trace of the failed problem-solving episode, the SBF model of the problem-solving process. Lex relies on the trace only. As a result its blame-assignment process is quite exhaustive, since it does not have any knowledge on the basis of which to localize its search.

4. As a result of its limited adaptation competence, Lex is able of improving its problem-solving efficiency only.

semantics

A very interesting feature of Lex, which AUTOGNOSTIC lacks, is its ability to generate training problems for itself. Lex's problem generator has several strategies that it can use to generate such pedagogical experiments. For example, it selects an operator with unrefined selection heuristics, and generates an integral that matches only some of the patterns in the operator condition. If the problem solver solves this problem, then the generalizer will be able to refine the operator's applicability conditions. Alternatively, the problem generator may take a problem already solved by Lex and tweak it slightly, in order to investigate whether the applicability conditions of the operators involved in its solution can be generalized. In general, these strategies rely on a hierarchical organization of the domain terms that are used in defining the selection heuristics. AUTOGNOSTIC does not require such an organization of the domain of its problem solvers, but its learning process would benefit if it could use such an organization in domains where it naturally exists. If AUTOGNOSTIC were extended with a problem-generating ability, it could potentially invoke this ability to gather evidence on which among the alternative concepts it has discovered should be integrated in the task structure.

Soar Soar [Laird *et al.* 1986, Laird *et al.* 1987, Rosenbloom *et al.* 1989] proposes the application of production rules as a universal problem-solving mechanism and chunking as a universal learning mechanism. Soar solves problems by search through problem spaces. A problem space has associated with it a set of states and a set of operators for moving through them. All the knowledge in Soar is encoded as productions. Given an initial state, all the relevant productions fire and when this activity stops, the production with the highest activation is applied. At this point, Soar may face the following kinds of impasses: no production can be chosen, two productions have exactly the same activation, and two productions have higher activation than each other. Associated with these impasses, Soar has subgoals that invoke search in corresponding problem spaces. When a subgoal generated by an impasse is accomplished its result is chunked into a new production rule indexed by the context in which the impasse occurred. Depending on the actual content of the production rules in a problem space Soar may exhibit an array of different problem-solving strategies. However, all these strategies arise from the same basic mechanism which does not make any commitments regarding the content of the production rules that should be included in a problem space. The chunking mechanism in Soar transforms productions that can be retrieved through a search in a separate problem space, called ps-retrievable productions, to productions that can be retrieved directly in a single activation cycle, called to k*-retrievable productions. Chunking is essentially a knowledge-compilation mechanism that increases the system's efficiency.

Chunking alone does not introduce new knowledge to the system. The way Soar can acquire new domain knowledge and new operators is by providing specifically tailored mechanisms for human users to declaratively specify new problem spaces. As a new problem space is developed, the chunking mechanism can be used to operationalize the rules of this space into immediately accessible productions. This method is similar to rote learning, where an external expert presents the knowledge to the system. Another way in which Soar can acquire new knowledge is through experimentation. For example, Soar may be told that the operators, i.e., their pre- and post-conditions, of a particular problem space are not completely defined, and can be switched in

experimentation mode. In that mode, Soar "tries" out the ill-defined operators, notices their results in its environment and from these results learns their pre- and post-conditions. This experimentation mechanism is a learning separate from chunking.

The first important difference between Soar and AUTOGNOSTIC is the level at which they analyze problem solving: Soar makes architectural commitments regarding the problem-solving process at a "syntactic" level of production rules, conditions of the rules activation of rules, and memory capacity. AUTOGNOSTIC analyzes problem solving at a "content" level in terms of tasks, methods and knowledge. Since the commitments of the two theories are at different levels of analysis, in principle, AUTOGNOSTIC's reflective problem-solving-and-learning process could be implemented in Soar. However, Soar would not give any guidance as to how to do that; AUTOGNOSTIC's theory of how to analyze and model problem solving would still be useful in such an endeavor.

Furthermore, Soar's impasses and AUTOGNOSTIC's failures are also quite different. Impasses are overt failures, i.e., the system reaches a state from which it does not know how to proceed. Soar does not require any feedback at all to recover from its impasses. It assumes that the knowledge necessary to address the impasse is available to the system, although in another problem space, and thus, it is not immediately accessible. If this assumption were violated, then Soar might reach an impasse from which it wouldn't be able to recover, or might run indefinitely searching for this non-available knowledge. On the other hand, AUTOGNOSTIC can use feedback from its environment, in the form of the desired solution. The feedback solution may or may not be already within the system's problem-solving competence and AUTOGNOSTIC is able to recognize whether the domain knowledge conveyed by the feedback is already available to it and whether its design elements are appropriately designed to produce it. If not, it can postulate what would be the necessary pieces of domain and/or functional knowledge that would lead to the production of the feedback. Thus, in AUTOGNOSTIC the acquisition of new knowledge is an integrated part of the problem-solving-and-learning cycle, since the system is able to recognize when such acquisition is needed.

The list below summarizes the comparison between Soar and AUTOGNOSTIC:

1. Soar has two separate learning mechanisms: chunking and experimentation. For the former it does not require feedback and for the latter the feedback it requires is the results of the application of each one of its operators. Soar does not recognize the need to modify its operators through problem solving; rather, it is told that its operators are not well-defined and then it proceeds to experiment with them in order to better define them. AUTOGNOSTIC on the other hand, independently recognizes the need to extend its domain knowledge and/or its task knowledge from the feedback it receives on its problem-solving behavior.

2. The learning tasks that Soar addresses are knowledge compilation, and functional knowledge acquisition. AUTOGNOSTIC's theory does not account for knowledge compilation, but it accounts for acquisition of both domain and functional knowledge in a single learning mechanism closely integrated with problem solving.

3. Soar's learning methods, i.e., chunking and experimentation, are quite both different from AUTOGNOSTIC's blame-assignment-and-repair learning method. Neither chunking nor experimentation involve blame assignment. In chunking the cause of the failure is always the fact that a rule is missing, and this rule is always discovered through search in another problem space. In experimentation, the error is in the operator with which Soar experiments, and the missing information can be found in the affects of this operator in the environment.

4. Chunking increases the problem-solving efficiency and experimentation may result in extended class of solvable problems, improved solution quality, and potentially improved efficiency also.

Soar's capability for experimentation could be a very useful extension in AUTOGNOSTIC's reasoning. In particular, in the cases where AUTOGNOSTIC notices that its comprehension of a task does not match with the actual behavior of the problem solver, if AUTOGNOSTIC had an experimentation mechanism, it could try out this task in several different contexts and learn

its actual information transformation from the produced solutions. Essentially, AUTOGNOSTIC in experimentation mode could repeatedly perform the ill-specified task until it has a consistent description of it.

Prodigy Prodigy
[Carbonell 1986, Carbonell *et al.* 1989, Minton 1990, Perez 1994, Veloso 1992] is a general architecture, consisting of a means-ends analysis non-linear, problem solver, an explanation-based learning element, a separate learning element for building abstraction hierarchies of operators, and an experimentation element for refining the pre- and post-conditions of its operators.

The explanation-based learning in Prodigy is used to learn control heuristics to select among the problem-solver's operators. It takes as input the trace of a problem-solving episode and produces as output control heuristics that the problem solver can use to better select among its operators. This is a task similar to Lex's learning task. Like in Lex, the trace of the problem-solving episode consists of a set of unsuccessful paths and a successful one. Unlike Lex however, where all operators in the unsuccessful path which do not belong in the successful one are incorrectly used, the blame assignment in Prodigy identifies a more restricted set of "interesting" choices to be credited with the success of the problem solver or to be blamed for the dead-end paths it pursued before: sole-alternative choices lead to selection heuristics, success choices lead to preference heuristics, failure choices lead to rejection heuristics, and choices that lead to paths with goal-interferences lead to preference heuristics.

Prodigy can acquire new knowledge, through its experimentation process. Like Soar, in experimentation mode, Prodigy performs actions that are not completely specified (i.e., operators with no well-defined postconditions) and from the results of these actions it learns their post-conditions. Thus, it improves its functional knowledge, by incrementally learning new operators. This is a separate process that does not interact with the normal problem-solving-and-learning cycle but is explicitly invoked by the human user of Prodigy.

Finally, Prodigy has yet another learning method, that is, the abstraction-hierarchy learning mechanism. This is an one-shot learning process which organizes the problem-solver's operator set in a hierarchy depending on the kinds of domain objects each operator applies to. Having a hierarchy of its operators, the problem solver prefers to apply the higher-level operators before the lower-level ones.

The list below summarizes the comparison between Prodigy and AUTOGNOSTIC:

1. The failure-driven learning method in Prodigy requires as feedback the preferred solution and the trace of its production. The trace can be input by an oracle, or Prodigy can produce it by exhaustively pursuing all the alternative paths until it reaches the preferred solution. In any case, it is assumed that the preferred solution is already within the competence of the problem solver. AUTOGNOSTIC's failure-driven learning requires the preferred solution only. AUTOGNOSTIC does not make the above assumption, and it admits that the parsing of the preferred solution may reveal gaps in its domain knowledge. In addition, because the problem solver in AUTOGNOSTIC may be hierarchical and therefore it is bot possible to infer the desired trace from the solution only, AUTOGNOSTIC needs the SBF model of the problem solver to infer (partially) this desired trace.

2. Prodigy proposes two separate methods for reorganizing the problem-solver's functional architecture (learning control heuristics, and abstraction hierarchy), and another one for acquiring new functional knowledge (experimentation). From these methods only one is invoked by the system itself upon the problem-solver's failure, the other two are invoked by an external agent. Thus, to a great extent, Prodigy does not decide by itself what it needs to learn. AUTOGNOSTIC, on the other hand, integrates these learning tasks in a single failure-driven learning process. This process independently decides which task should be invoked to address the failure.

3. Prodigy is equipped with three different learning mechanisms, only one of which is failure-driven. In that mechanism the blame-assignment task is accomplished through direct comparison of a failed and a successful reasoning path, like in Lex. Prodigy focuses the set of

elements on which it assigns the blame for its failure using a set of four predefined patterns of erroneous operator selection. These four patterns essentially define different types of cause of failure. This taxonomy is quite limited with respect to the types of causes that AUTOGNOSTIC's blame assignment can recognize, which range from domain-knowledge errors, to task-structure organization errors to task functionality errors. Furthermore, AUTOGNOSTIC's learning strategy relies on its SBF model of the problem-solving process rather than on the correct reasoning trace.

4. Each of the different learning processes in Prodigy may result in different types of performance improvement. Learning control heuristics, for example, may lead to improved efficiency and also to improved solution quality. Learning an abstraction hierarchy of the problem-solving operators also results in improved efficiency. Finally, experimentation may result in extended class of solvable problems.

Currently, AUTOGNOSTIC's learning theory assumes the existence of a task model of the problem-solver's reasoning. This task model is hierarchically organized and at the lowest level of the organization there are the elementary tasks of the problem solver which can be accomplished by its operators. AUTOGNOSTIC could use a method similar to Prodigy's method for hierarchically organizing the problem-solver's operators to build its task model given only the elementary tasks that the problem solver can perform. Although Prodigy's method would be useful in creating the task hierarchy, but it would not suffice. This is because AUTOGNOSTIC's SBF model requires the specification of the functional semantics of the tasks at all levels, and Prodigy completely lacks such knowledge.

Hacker and Chef Hacker [Sussman 1975] was a precursor of case-based reasoning. It has a memory of abstract plans, which it instantiates to solve the problems it is presented with. Each plan specifies a set of subgoals that are necessary to be accomplished for the problem at hand to be solved, and a sequence of steps for solving the problem. In addition, it explains how each step relates to the accomplishment of one of the plan's subgoals. In this limited sense, Hacker has knowledge about the functional and compositional semantics of the elements of its abstract plans. The specification of each problem presented to Hacker consists of a set of goals that need to be achieved. Hacker instantiates a (set of) abstract plans to cover the complete set of the goals in the problem at hand. Hacker's planner makes the linearity assumption, that is, it plans for each problem goal separately and it assumes that the resulting plans can be concatenated to each other and that they will not interfere with each other. During planning, in addition to keeping a trace of its problem solving, it also keeps a record of the goal decomposition which results in this trace.

When Hacker has produced a plan for a given problem, it proceeds to simulate its execution. If the plan execution fails, then it proceeds to identify the cause of the failure. Hacker addresses a relatively narrow class of errors, namely the violation of its linearity assumption. Hacker is equipped with a set of patterns of pathological goal interactions, which may lead to such violations. Given a failed plan, Hacker matches its failure patterns to the goal-structure and trace of the failed plan. Having identified which of these patterns applies to its current failure, it subsequently modifies the failed plan according to the fixes associated with this pattern.

After successfully modifying the failed plan, Hacker generalizes it and stores it in its memory of abstract plans for future reuse. It also generates a critic, a specialization of the pathological goal interaction in the current context so that its planner can use it to anticipate similar potential errors.

Chef [Hammond 1987, Hammond 1989] uses the same planning process as Hacker, only instead of using abstract plans which can be instantiated in different situations, it uses *cases*, i.e., specific plans, which it adapts to new problem situations. The learning element in Chef takes as input the trace of the plan's simulation, i.e., the affects of each individual plan step to the state of the world, as inferred by the simulator. It evaluates the difference between the expected state of the world and the actual state after the simulated execution of the plan, and it identifies the plan step(s) that are responsible for the failure. Then, it modifies the failed plan according to a set of repair plans available to it.

Chef and AUTOGNOSTIC address a slightly different learning task. Chef identifies what in the planning process caused an undesired aspect in the state of the world after the execution of the plan.

AUTOGNOSTIC, on the other hand, identifies what in the problem-solving process could be changed in order to achieve a desired state that was not achieved by the execution of the solution. Chef uses the simulation trace to identify where in the plan execution the undesired affects first appear. AUTOGNOSTIC, on the other hand, uses its SBF model of the problem solver to construct a possible alternative trace which could lead to the desired-but-unaccomplished solution.

The list below summarizes the comparison between the Hacker and Chef on one hand and AUTOGNOSTIC on the other.

1. The conditions that signify failure to Hacker and Chef are different from AUTOGNOSTIC's failure conditions. Hacker and Chef recognize that the planner has failed when the produced plan cannot be successfully simulated in the world. AUTOGNOSTIC recognizes that the problem solver has failed when the environment presents with an alternative solution that would have been preferable to the one that the problem solver actually produced.

2. The theories of Chef and AUTOGNOSTIC admit the need for a wider range of learning tasks than Hacker's learning theory. Hacker addresses errors caused by the linearity assumption and essentially, it only reorganizes the goal structure of the failed plan. Chef on the other hand, in addition to reordering steps in a plan, it can also suggest the substitution of a step with another. However, Chef can only use the steps it already knows about and, unlike AUTOGNOSTIC, it cannot recognize the need for new steps or postulate such new steps.

3. Due to the difference in the overall learning problem they address, the blame-assignment processes of these three theories are also quite different. The trace of the plan simulation enables Hacker and Chef to localize the fault to this part of the plan which occurred before the failure. AUTOGNOSTIC uses the SBF model of the problem solver to localize the error in that element whose information transformation incompatible with the desired solution. All three systems index repair strategies from the types of failure causes that they can identify.

4. Both Hacker and Chef perform some kind of knowledge compilation, that is they both transfer knowledge that they have in their "learning" modules, i.e., the modules responsible for blame assignment and repair, to their planning modules. Thus they improve their efficiency but do not change the quality of their solutions or the set of problems they can address.

Celia Celia [Redmond 1992] is a apprentice system that learns by watching an expert performing a task, such as troubleshooting a car. Celia does not only watch the expert, it also tries to predict the expert's actions and solve the problem at hand itself. Celia begins with an incomplete and possibly buggy set of diagnostic actions it can perform. Furthermore it does not know all the possible ways in which to combine these actions. Thus its expectations may fail in several different ways. For example, the expert may perform an action unknown to Celia, or he/she may combine known actions in a novel manner. When its expectations regarding the next action of the expert fail, Celia recognizes an opportunity to learn. Celia compares the action it expected with the actual action of the expert and uses the difference between the two as an index into a complete model of its car domain. This fault model is annotated with diagnostic actions relevant to each part of the car, and these annotations enable Celia to infer the expert's goals and thus integrate the expert's action into its own diagnostic process. Celia learns new actions, and new diagnostic procedures, parts of which are annotated with subgoals of the overall diagnostic task.

To summarize the comparison between Celia and AUTOGNOSTIC:

1. Celia learns in an apprentice mode, by attempting to solve a problem and at the same time looking at the actions of an expert. In that mode, Celia has step-by-step feedback on its actions. AUTOGNOSTIC learns by reflecting on its a complete problem-solving episode having as feedback only the desired solution to the overall problem. The detection of failure in Celia is possible through direct comparison of the system's and the expert's actions, and when such a failure is detected, Celia has available to it the trace of its own failed behavior and the trace of the expert's reasoning.

2. Celia's apprenticeship enables to extend its diagnostic goal structure. Essentially it learns which particular diagnostic actions are useful under which conditions. This task is, to some extend, similar to AUTOGNOSTIC's learning of new tasks only Celia does not learn new tasks, i.e., how to accomplish novel information transformations, but rather specific diagnostic actions.

3. Celia receives a step-by-step feedback on the quality of its predictions, which eliminates the problem of the localization of the failure. On the other hand, AUTOGNOSTIC receives feedback only at the end of its problem solving, and at that point it spawns a blame- assignment task to address the problem of localization. For the subsequent repair, Celia assumes a fault domain model which always explains the expert's actions and enables it to integrate them in its own diagnostic goal structure. On the other hand, AUTOGNOSTIC has several different repair plans each one appropriate for different types of learning tasks.

4. Celia extends the population of problems it can solve and the quality of the solutions it produces.

The learning theories of Celia and AUTOGNOSTIC are at different stages of the development of an intelligent problem solver and learner. Celia works at the novice level, where the agent does not have a "theory about how it should solve problems". AUTOGNOSTIC, on the other hand, assumes that the agent has already formulated an abstract theory of its own problem solving. Thus in general, the conditions of their applicability do not overlap. However, it is interesting to note that Celia is performing a learning task which could potentially provide AUTOGNOSTIC with enough information to generate the SBF model of its problem solving. To date, AUTOGNOSTIC's theory does not provide an account of how this theory of problem solving might be developed. One way that AUTOGNOSTIC might be able to learn the SBF model, would be by induction of Celia's plans. For example, it could potentially induce semantic relations over a sequence of similar diagnostic steps in plans with similar goal-subgoal structures. It could also induce method-selection conditions over a sequence of plans which accomplish similar goals although through different action sequences.

IDeAL IDeAL [Bhatta 1995] is a design and learning system. It performs analogical conceptual design. IDeAL builds on the Kritik system. It is equipped with a memory of physical designs along with their SBF models, indexed by their functional specifications. Given the functional specification of a new desired design, it retrieves from its memory of past designs one that delivers a function similar to the desired one. IDeAL is able to perform both intra- and cross-domain analogies, that is, it can reuse past designs to design new ones not only in the same domain as the old design but also in other domains. Its cross-domain analogical process is mediated by abstractions of general principles and engineering mechanisms, which IDeAL learns from its own design problem-solving experiences.

The process by which IDeAL learns its abstractions is failure-driven. When IDeAL is presented with a design problem which it cannot solve with its current knowledge, it receives as feedback from the world the desired design. Then it compares the desired design to the past design which it tried to adapt, and identifies the parts of their internal causal behaviors which are common across the two. At this point, it infers that the commonalities across their delivered functions result through the commonalities in their internal behaviors, and stores in its memory the abstract function-behavior package. In the future, where similar functionalities are desired and the specific designs in IDeAL's memory are not useful, IDeAL can retrieve and instantiate this abstract function-behavior mechanism.

There are several interesting themes that run common in AUTOGNOSTIC and IDeAL: Both systems perform design: AUTOGNOSTIC of abstract devices, IDeAL of physical ones. Both have deep functional and causal knowledge of the artifacts they design, which is expressed in the SBF modeling framework. IDeAL's process for learning engineering mechanisms is based on this deep knowledge, much like AUTOGNOSTIC's learning process is. In fact, IDeAL's engineering mechanisms enable it to solve new classes of design-adaptation problems, just like AUTOGNOSTIC's new tasks enable the problem-solver to draw new classes of inferences. IDeAL's learning process however is tailored for design problem solving, that is, it only learns new mechanisms that its

designer can use, where AUTOGNOSTIC's learning is not particular to a problem-solving task. For that reason, IDeAL does not have, and it does not need, explicit meta-level descriptions of either the problem-solver's domain theory or task structure, since, the problem solver and the learner in IDeAL reason about the same domain using the same knowledge.

To summarize the comparison between IDeAL and AUTOGNOSTIC, let us draw the following list:

1. Both systems embody theories of model- based adaptation of teleological artifacts, physical in the case of IDeAL, abstract in the case of AUTOGNOSTIC. Both theories assume models that capture the functional semantics of the design elements of which the teleological artifact consists, and the compositional semantics of the interactions of these elements. Finally, both theories require external feedback, although IDeAL can use a wider range of types of feedback than AUTOGNOSTIC.

2. IDeAL's learning of abstract engineering mechanisms is similar to AUTOGNOSTIC's learning of new tasks in order to introduce new classes of information-transformations in a failing task structure. IDeAL uses an abstract engineering mechanism as a mechanism for adapting old designs. In that sense, a new engineering mechanism is a new class of adaptation inferences. Essentially, IDeAL learns different instances of the adaptation task.

3. AUTOGNOSTIC's process for discovering new design elements that could enable the failing artifact to deliver the desired behavior, is domain and task independent and is mediated by its explicit meta-level understanding of the domain of the artifact. The corresponding process in IDeAL is specifically tailored for the domain of physical devices.

4. IDeAL's learning of engineering mechanisms enable it to extend the class of design problems it can solve, and also to potentially improve the quality of the solutions it produces.

10.3 AI Reflective Systems

AI research on meta-reasoning can be broadly categorized in two classes: this research that focuses on the affects of meta-reasoning to deliberative and thus flexible intelligent problem-solving activity [Goel *et al.* 1995, Hayes-Roth and Hayes-Roth 1978, Kuokka 1990, Stefik 1981, Turner 1989, Wilensky 1984] and the work that focuses on the affects of reflection on recovery from failure and learning. Because AUTOGNOSTIC's theory investigates mostly the affects of reflection on learning, this section focuses on the latter body of research. In this context, there are again interactive reflective systems [Davis 1977, Weintraub 1991], that focus on the tasks of supporting the knowledge acquisition process, and reflective autonomous systems [Mitchell *et al.* 1989, Freed *et al.* 1992, Ram and Cox 1994].

All reflective intelligent systems share a common characteristic, i.e., the fact that they all have a model of their problem solving. Thus, the two dimensions along which it is interesting to compare different reflective systems are first, the aspects of the problem solving that their model captures, and second, how this model is used in learning.

Teiresias Teiresias [Davis 1977] is an interactive tool aimed to support the knowledge-acquisition process for Mycin. Mycin is an expert system that diagnoses infectious bacterial diseases, using backward chaining of production rules [Shortliffe 1976]. Teiresias, has a model of the knowledge that Mycin uses for its reasoning, in terms of *meta-rules*. Teiresias' meta-rules organize the rules of Mycin as positive and negative evidence for a hierarchy of possible diseases. All the knowledge in Mycin, i.e., domain knowledge regarding the diseases that Mycin diagnoses, and functional knowledge regarding how to carry out the diagnostic process, is represented as rules, and Mycin's processing is a repetitive sequence of asserting or negating propositions based on the rules that fire at each step. Thus, Teiresias meta-rule model captures, to some extent, the interactions between the problem-solver's design elements. Furthermore, in some sense, it also captures the compositional

semantics of Mycin's design elements, because the backward chaining process implies that each new rule modifies the set of currently held propositions by negating (deleting) a proposition in the set, or affirming (adding) a new one. Note, that the propositions asserted or negated may refer to diseases or to the diagnostic process itself, i.e., there is no difference between rules that refer to the domain of diseases and rules that refer to strategic diagnostic knowledge. Thus, it cannot capture any functional semantics of the kinds of inferences that each element draws, and the role it plays in the diagnostic process.

When Mycin produces an erroneous answer, Teiresias presents the trace of its reasoning to a domain expert, beginning from the most recent rule that was used. As it presents the rule upon which Mycin's inferencing was based, Teiresias asks the expert to evaluate (a) whether the rule is correct, (b) whether the situation on which the rule applied is correct, or (c) whether there is another rule that could have rejected the erroneous conclusion. If the rule is incorrect, Teiresias prompts the expert to modify it. If the situation in which it was applied is incorrect, Teiresias prompts the expert to specify what the situation, i.e., the set of propositions held by Mycin, should have been before the rule firing. Finally, if there is a rule that could have negated the erroneous conclusion, Teiresias asks the expert to specify it.

Teiresias itself does not learn. It simply guides the refinement of Mycin's knowledge base by the expert. It presents the expert with "rule templates" which the expert fills and thus introduces new knowledge in the system.

Teiresias model of Mycin's reasoning does not differentiate between domain and functional knowledge. Thus rules regarding symptoms of diseases, for example, and rules regarding steps of the diagnostic process are treated the same. Thus the introduction of a new rule may imply the introduction of any of the above kinds of knowledge. On the other hand, AUTOGNOSTIC's SBF model of the problem solver distinguishes its design elements with respect to their content. That is, it distinguishes its functional elements, tasks, from the rules of their composition, methods, and from its domain knowledge, and it uses a different set of terms to specify each one of them.

Irrespective of their different models of the problem-solver, the blame-assignment processes of Teiresias and AUTOGNOSTIC share many commonalities. They both investigate similar possible causes of error, "ask similar questions", in the problem-solver's reasoning. AUTOGNOSTIC evaluates whether each step matches the semantics of the task that produced it (similar to Teiresias' question a), or whether some part of the input should have been different (similar to Teiresias' question b), or otherwise it suggests as a possible modification the falsification of the domain knowledge that make the problem-solver's inference possible (similar to Teiresias' question c). However, instead of having a resident expert evaluating the system's inferences like Teiresias, AUTOGNOSTIC uses the task semantics that capture the desired behavior expected of each task, to evaluate whether the actual behavior meets these expectations. It is the same knowledge that enables AUTOGNOSTIC to infer correct alternatives to the actual failed inferences of the problem solver, instead of Teiresias relying on the expert to provide the correct rules.

The list below summarizes the comparison between Teiresias and AUTOGNOSTIC:

1. Both systems have a model of the problem-solving process, only they analyze problem solving at two very different levels: Teiresias at the level of rule-based problem solving, in terms of rules asserting or negating facts, and AUTOGNOSTIC at the level of tasks recursively decomposed by methods into elementary tasks which draw inferences based on domain knowledge.

2. (a) Teiresias does not have any specification of what the problem-solving elements are meant to do. Thus it depends on an external agent to evaluate the correctness of its inferences. AUTOGNOSTIC, on the other hand, assigns the blame for its failure autonomously. It explicitly represents the task semantics, and thus, it is able to evaluate particular inferences with respect to whether or not they meet the semantics of the tasks in service of which they were drawn.

   (b) Because of the same reason, Teiresias is also unable to specify what to learn, but only at the very "syntactic" level of "a rule asserting that ...". AUTOGNOSTIC on the other hand, is able to specify the class of inferences desired of the problem solver.

Cream Cream [Weintraub 1991] models knowledge-based expert systems in terms of the generic tasks they accomplish. Cream was tested with a diagnostic expert system, Qwads, which performs gait analysis as an instance of the *hypothesis-assembly* generic task. Cream has a task-structure model of the Qwads diagnostic task. For each task, the model specifies the method used to accomplish it and the subtasks this method sets up. In addition, for each task, Cream's model specifies the potential errors that can occur while it is accomplished which can lead to the failure of the overall diagnosis.

When Qwads fails, i.e., when its answer differs from the answer of the domain expert, Cream proceeds to identify a set of tasks which could be at fault. The basic idea in Cream's blame assignment is that, somewhere in its reasoning, Qwads preferred its own erroneous answer instead of the expert's. Cream's goal is to identify where. Cream inspects each task and evaluates whether any of the candidate errors with which it is annotated can account for this task's failure to lead to the desired answer. At the end, Cream proposes a set of tasks along with candidate errors for each one of them that can potentially explain why Qwads failed to produce the desired answer.

Cream addresses only the blame-assignment subtask of the overall failure-driven learning problem. As such, Cream's task is very similar to Teiresias task, but where Teiresias relies on the expert to identify errors and/or gaps in the knowledge base, Cream does so in an autonomous manner, based on the comprehension of the function of the knowledge-base design elements as captured in their description by generic tasks.

Cream's blame-assignment process requires a "fault model" of the reasoning of the knowledge-based system, i.e., an analysis of the problems that can potentially occur with each task (similar to side effects of device components). If this model, is incomplete or incorrect, Cream cannot correct it, in fact, Cream cannot even recognize the inadequacy of its model. In comparison, AUTOGNOSTIC requires only a model of the system's correct behavior, and it is able to incrementally improve and correct any such model it is provided with. Furthermore, Cream does not autonomously modify the knowledge-based system, but depends on a domain expert to do so.

The list below summarizes the comparison between Cream and AUTOGNOSTIC.

1. Both theories assume a task-structure model of the problem solver which they reflect upon. Cream's models differ from AUTOGNOSTIC's in two important respects however:

   (a) Cream's model is annotated with the "typical faults" in the subtasks that the system's accomplishes. It is essentially a fault model of the problem solver. AUTOGNOSTIC's SBF model, on the other hand, specifies the functional semantics, i.e., the expected correct behavior of each one of the system's subtasks.

   (b) Furthermore, Cream's model assumes a single-strategy problem solver where AUTOGNOSTIC's SBF model allows for multi-strategy problem solving.

2. Cream uses its model for the diagnosis of the failing problem solver, it does not actually modify the failing system. AUTOGNOSTIC, on the other hand, in addition to identifying the potential causes for the problem-solver's failure, selects which among them to address and actually repairs it.

Theo Theo [Mitchell *et al.* 1989] is a frame-based problem-solving and learning architecture. In Theo, knowledge is represented in frames. In Theo, problems are also entities represented as frames organized in is-a hierarchies, just like knowledge. Theo is a reflective architecture, in that, it can have beliefs about its knowledge and about the problems it solves and the methods it has to address them. That is, Theo does not provide a content theory about the types of functional elements that give rise to problem-solving behavior, but rather, it implements reflection through a uniform representation scheme.

Problem solving in Theo is equivalent to inferring slot values. A problem is specified as an <*entity*> <*slot*> pair and Theo's goal is to produce a <*value*> as an answer. When it produces an answer, it updates the frame of the <*entity*> with it. This is one type of learning that Theo is able to accomplish, i.e., caching. In addition, it stores the sequence of all the frame slots it had to access in

order to produce the solution, as a method for this type of problem at the most general frame where it is applicable. This is another type of learning in Theo which results in macro-operators. This sequence is essentially an explanation of how the solution was produced. This explanation also enables a kind of truth maintenance in Theo's knowledge, since it enables it to "erase" the values it has produced when some of the slots on which their production depends is modified. Finally, Theo keeps statistics on the success and failures of the different methods it may have for solving some type of problem (i.e., producing the value of a slot) and uses induction to order these methods, and learn preference heuristics among them. None of these three learning methods is driven from failure. They all are based on Theo's successful problem solving. Furthermore, Theo, does not get any feedback on its problem solving. Thus, the learning problem it addresses is very different from the one addressed by AUTOGNOSTIC.

The list below summarizes the comparison between Theo and AUTOGNOSTIC:

1. Theo and AUTOGNOSTIC adopt a very different view towards reflection. In Theo reflection is viewed as the ability of an agent to hold beliefs about its own knowledge and reasoning, and is implemented through a uniform representation of knowledge, problems and beliefs. This is a different approach to AUTOGNOSTIC's reflection process which commits to a content theory of problem solving, rather to a representation scheme. In AUTOGNOSTIC reflection is investigated in the specific context of failure-driven learning, and this is why it focuses on a specific kind of "beliefs" i.e., the agent's beliefs regarding the functions of its own design elements, and their composition into its overall behavior.

2. Theo's learning process is not failure- but rather success-driven. The learning tasks that they accomplish are different: solution cashing, operator composition, and method-selection criteria in the case of Theo, domain-knowledge acquisition and reorganization, and task-structure modifications in the case of AUTOGNOSTIC.

Meta-Aqua and Meta-TS Meta-Aqua [Ram and Cox 1994] and Meta-TS [Ram *et al.* 1993] are two different failure-driven learning systems which investigate the same theory of reflective learning in the domains of story understanding and troubleshooting respectively.

Meta-Aqua operates in the story-understanding domain. The story understander in Meta-Aqua uses cases of stories it has seen in the past and explanation patterns (XPs) to interpret the stories it reads and draw inferences from them. While reading a particular story, it records its reasoning in terms of four basic types of inferences, called Trace Meta-XPs (trace meta-level explanation patterns). It recognizes failures when the input from the story is in direct conflict with some of the understander's expectations, as they were set up by its cases and explanations. To assign blame for these failures Meta-Aqua uses Introspective Meta-XPs (introspective meta-level explanation patterns). Just as explanation patterns are causal patterns explaining the interactions among a set of domain concepts, meta-level explanation patterns are causal patterns explaining the interactions among the different steps of the story-understander's reasoning. These patterns can be thought of as fault models, i.e., prototypical patterns of erroneous interactions among reasoning steps as opposed to AUTOGNOSTIC's model of the "correct" reasoning process.

In Meta-Aqua the problem solver has beliefs and goals. Goals drive much of the understanding process, but also the learning process in Meta-Aqua. When one of its expectations has failed, Meta-Aqua attempts to match its meta-xps with the trace of the failed reasoning episode. When a meta-xp has been identified in the failed trace, it posts a knowledge goal [Ram and Hunter 1992]. A knowledge goal consists of a specification of the knowledge adaptation (acquisition, revision, and reorganization) that needs to occur, and a specification of why it is needed. Given a set of knowledge goals, and a set of learning strategies which can potentially be applied to eliminate them (each Meta-XP indexes an array of strategies that can be potentially applicable to it), Meta-Aqua employs a NONLIN-like planner to decide how its knowledge goals can be better met.

The blame-assignment method of Meta-Aqua is fundamentally different from that of AUTOGNOSTIC. To a great extent, this difference arises from the kinds of tasks these theories deal with. Meta-Aqua's task is essentially abductive, driven mainly by the input data, without a well defined task structure. On the other hand, the tasks of the problem solvers that AUTOGNOSTIC has been

integrated with, i.e., planning and design, have a stronger internal task structure. Comprehension tasks are usually thought of as search in a hypothesis space. Even though the understander may have goals that guide this search, this guidance is usually less strong than the processes of tasks such as planning and design. Thus, in the former kind of tasks the assumption of "typical faults" is more plausible than the assumption of "understanding how the task is meant to be accomplished". It is interesting to note here that Cream [Weintraub 1991] also uses a fault model for assigning blame for the failures of a similar task, i.e., diagnosis. On the other hand, in the case of tasks such as planning and design, the problem solver usually has very complex internal processes that employ multiple strategies which interact in various although usually well-formulated ways, which, are synthesized from well-formulated elementary tasks. Thus, although because of the complexity of the interactions the exhaustive pre-compilation of all the types of errors that can occur is too difficult, the specification of how the process is designed to work in the first place is easier. Furthermore, because the description of the typical errors is in terms of the current problem-solver's tasks, a blame-assignment process that relies on them cannot identify errors in the formulation of these tasks or postulate the need for new ones. Thus, Meta-Aqua cannot modify its own task structure. However, because AUTOGNOSTIC's theory relies on an abstract comprehension of the design of the problem solver, AUTOGNOSTIC can find and fix errors in the problem-solver's task structures.

With respect to the types of performance improvement that Meta-Aqua is able to accomplish, as it acquires more knowledge and as it reorganizes its knowledge better, it extends the population of stories it can understand. Due to the nature of its task, there are no well- defined metrics for efficiency of the story understanding process or quality of its output.

The following list summarizes the comparison between Meta-Aqua and Meta-TS on one hand, and AUTOGNOSTIC on the other:

1. Reflection in Meta-Aqua and Meta-TS is based on a set of Meta-XPs. Trace Meta-XPs are used to record each problem solving episode in terms of a repeated cycle of four basic steps. Introspective Meta-XPs, which are also expressed in the same terms, explain a-priori known faulty interactions among these basic tasks, and they are essentially pre-compiled models explaining typical problem-solving failures. AUTOGNOSTIC, on the other hand, uses its abstract comprehension of the problem solver in terms of its SBF model to interpret its reasoning steps, and identify their problematic interactions in case of failure.

2. The blame-assignment process is based on the matching of the Introspective Meta-XPs against the record of the story- understanding (troubleshooting in the case of Meta-TS) episode which is expressed in terms of Trace Meta-XPs. Unlike generic tasks however, trace Meta-XPs do not capture any knowledge regarding the functional semantics of these steps. In AUTOGNOSTIC, on the other hand, the erroneous interactions among the reasoning steps are not known a-priori. Instead, they are inferred in the context of each particular episode, based on a comparison between what the functional role of these steps should be, as captured in the SBF model of the problem solver, and what their actual behavior was.

An interesting aspect of Meta-Aqua is its ability to post several knowledge goals, in response to each failure, and explicitly plan on how to achieve them. AUTOGNOSTIC's theory admits the possibility that many learning tasks can be potentially relevant to eliminating the cause of a particular failure, and it uses a set of heuristics to decide which one among them to accomplish. The need for a more sophisticated planning process in dealing with multiple possible learning tasks did not arise in the evaluation of AUTOGNOSTIC up to date. However, it could be the case that with less-detailed SBF models for its problem solvers, many more potentially useful learning tasks would be identified, and AUTOGNOSTIC's heuristics could be proven insufficient. Such cases might give rise to the need for a more complex planning process.

Castle Castle [Freed *et al.* 1992] is a reflective learning system implemented in the domain of chess playing. Castle reflects on its failures, i.e., its unsuccessful moves, in order to learn how to better anticipate the consequences of its actions. Castle has a model of its game-playing process in terms of the basic reasoning steps it takes to decide its next move, such as threat detection, potential move generation, and move selection.

Castle views problem solving as a behavior arising from the interactions of the problem-solver's *components*. For each one of these components Castle has a description of its correct performance, and the assumptions that underlie its correct performance and its effectiveness in the overall problem solving.

Both of these annotations together capture the functional semantics of the components, in a way similar to the way AUTOGNOSTIC captures the functional semantics of the problem-solver's tasks. However, there are some important differences between Castle's modeling of problem solving and AUTOGNOSTIC's. For example, in Castle the problem solver is assumed to be non-hierarchical (the problem solver consists of a sequence of components), it uses a single strategy (the problem solver is a single sequence of components), and it is non-recursive (it lacks the concept of prototypical tasks which can be performed in service of different subgoals in the task structure). These assumptions limit the range of problem-solving processes that Castle can describe and reflect on, and have some important implications to Castle's reflection process. The lack of an organization among the problem-solver's components (more sophisticated than simple sequencing) would, in general, increase the complexity of the blame-assignment process if the number of components for a particular problem solver were large. In the case of a complex problem-solver the sequence of its components would be rather long. The blame- assignment process would potentially evaluate each one of these components to localize the cause of the failure and that would be computationally expensive. In contrast, AUTOGNOSTIC has a hierarchical description of the problem solver, and it reasons only about a little number of tasks at each level. Furthermore, the lack of any compositional semantics precludes Castle from reasoning explicitly about the consequences of the modifications it proposes to the problem solver. As a result it cannot explicitly reason about how to maintain the consistency of the problem-solving process after each modification. In contrast, AUTOGNOSTIC uses its knowledge regarding the composition of the problems-solver's design elements, as captured in the specification of methods in the SBF model, to reason about the affects of each adaptation it performs and thus maintain the consistency of the problem solving.

Castle's learning results in improvement of the quality of its moves. The notions of process efficiency and population of problems are not possible to define in Castle's domain.

The list below summarizes the comparison between Castle and AUTOGNOSTIC:

1. Both Castle's and AUTOGNOSTIC's models of the problem solver capture its functional and the compositional semantics. However, due to its very limited range of composition rules, Castle models and reasons about single-strategy, non-hierarchical problem solvers, where AUTOGNOSTIC can deal also with multi-strategy, hierarchical problem solving.

2. Both theories use their models of the problem solving in similar ways. They both evaluate the actual behavior of the problem- solver's reasoning steps against their functional specification. Next, they use their knowledge about the composition of these steps to trace the cause of the failure backward, towards steps as early in the process as possible. However, in addition to the functional semantics, AUTOGNOSTIC also models the semantics of the recursive composition of these basic design elements into higher-level tasks and finally into the overall task of the problem solver. This knowledge enables AUTOGNOSTIC to reason about how to maintain the consistency of the problem solver while modifying it. Castle's compositional semantics are very limited.

10.4 Psychological Research on Reflection

Research in psychology [Chi 1987, Weinart 1987] also has been investigating the issue of meta-cognition and its affects on performance. The term *meta-cognition* refers to the deep understanding of knowledge which can be manifested either as effective use of that knowledge, or as overt description of the knowledge in question. Meta-cognitive behavior is difficult to discern in human activities, for two reasons [Brown 1987]:

1. because a particular action can be either in support of either a cognitive or a meta-cognitive function; for example, asking questions on a particular piece of text can be an action resulting

from an effort to improving the understanding of the reader, or alternatively, from his/her effort to monitor its understanding progress, and

2. because the same term has been used to refer both to knowledge about cognition and also to regulation of cognition, supported by such meta-knowledge.

In general, however, planning activities, monitoring activities, and the evaluation of activity outcomes are considered as meta-cognitive activities.

Early work on meta-cognition aimed to investigate why children do not use spontaneously effective strategies in memorizing, although they are able to use them when prompted by adults. Flavell [Flavell 1971] suggested that children are not aware that they need to use particular strategies in particular situations. Along similar lines, Baker and Brown [Baker and Brown 1981] said that "if the child is not aware of his own limitations as a learner or the complexity of the task at hand, then he can hardly be expected to take preventive actions, in order to maximize performance".

Early on, children's development during middle childhood was interpreted as the formation of self-regulatory skills. According to Piaget [1971] "knowledge organizes and regulates information by means of auto-control systems directed toward adaptation, in other words, toward the solving of its problems". Piaget [1976] distinguished self-regulatory activities in terms of the degree to which they are consciously accessible to the agent. He proposed three levels of self-regulation: *autonomous*, which involves unconscious adjustments regulating, fine-tuning, and modulating behavior, *active*, which involves constructing and testing mental theories of action in parallel with action, and *conscious*, which involves the mental construction, simulation, and testing of competence theories. Mature agents exhibit the third kind of reflection, i.e., conscious. AUTOGNOSTIC's reflection process is consistent with the properties of self-regulation at the conscious level, with the exception that it does require some kind of trigger which can usually be provided through action. The SBF model of problem solving is supports mental simulation, and enables the agent to build new competence theories by consistently modifying its current one.

Mischel [1981] views children as sophisticated psychologists who develop and use psychological principles to understand social behavior, their own conduct and to achieve increasing control over their environments.

Kluwe [1982] defines as meta-cognitive activities, activities where the agent has knowledge about his own thinking and that of other agents and also monitors and regulates the course of its own thinking. According to Kluwe, the usefulness of such a self-regulatory mechanism lies in several reasons. First, the agent has to meet varying cognitive demands and thus its own thinking has to be regulated accordingly. Second, reasoning about a problem is non- deterministic, since in general, there are several different ways to solve it. Thus the agent has to be able to make executive decisions as to which course of reasoning to follow. Finally, human information processing can always be improved, and this requires monitoring and careful evaluation of one's own thinking.

Karmiloff-Smith [1979] attributed to meta-cognition the U-shaped curve of young children performance. She proposed that early on, success is the result of "exhaustive knowledge" of competent performance; essentially, in the beginning, children attempt each task as a unique individual problem, and their problem-solving knowledge is associated with specific problems. Soon enough however, they notice regularities among problems and the ways in which they can be solved, and the most dominant regularity becomes their current "competence theory". This "competence theory" formulation has as a result the deterioration of their performance since the most dominant regularity does not cover all problem instances, and thus failures arise as exceptions/errors which are ignored in the beginning. As the population of exceptions increases, the most dominant regularity in the exception population becomes a competence theory for solving the class of problems that is identified with the exceptions. Finally, the two competence theories are identified into one, which involves the generation of a global rule for "why things work" covering both theories. AUTOGNOSTIC's reflection process shares this goal of unifying the exceptions (the failures) with the current competence theory (the task structure) however, it takes a more greedy approach to this end: namely, it tries to modify its task structure with each failure in order to unify this problem with the class of problems that this task structure can correctly solve.

Catrambone [1993a,1993b] investigated whether or not explicit "labeling" of the reasoning steps in the examples presented to students would help. He found that providing extra labels

for the intermediate steps enabled students to associate subgoals with these labels, and in some sense resulted in the development of a goal structure. In turn, this goal structure enabled them to solve novel problems which involved the same basic subgoals but for which the particular method (formula) they were taught in the examples would not be applicable. This notion of *labels been associated with subgoals* closely matches the knowledge SBF models capture about problem-solving subtasks: each subtask in the SBF model of a problem-solver's reasoning specifies a subgoal in the course of the overall problem solving, and it is associated with the types of information that it produces for output. These output information types seem equivalent to the labels provided in this experiment.

Finally, Pirolli and Recker [1992] investigated the affects of reflection to transfer in the context of learning how to program recursive functions in lisp. They gave their subjects a set of number-recursion problems (i.e., factorial) and list-recursion problems (i.e., list intersection). They reported that their subjects formulated *abstractions* over several problem instances. Further, they modified their own abstractions by unifying and discriminating them based on similarities or differences over problem instances. In the examples we discussed in this paper, instances of both these types of abstractions can be found in AUTOGNOSTIC's reflection process. The process of modifying the task structure of the instance task is a discriminating abstraction: it distinguishes the instance from the previously unifying abstraction of the prototypical task, based on the requirements of a particular problem. The process of abstracting the modified task structure again at the level of the prototypical task, once all instances have been modified in a similar manner, is a unification abstraction.

10.5 Design and Device Modeling

This thesis views reflective, failure-driven learning as redesign of an imperfect problem-solving task structure. Thus it builds on a line of earlier work on design, and more specifically on adaptive design [Barletta and Mark 1988, Hinrichs 1989, Mostow 1990, Sycara and Navinchandra 1989]. The process of design adaptation in AUTOGNOSTIC is supported by the model of the device under inspection. Thus, it shares a lot of common aspects with model- based reasoning systems [Bylander and Chandrasekaran 1985], [Davis 1984], [de Kleer and Brown 1984], [Forbus 1984], [Goel 1989], [Kuipers 1986], [Sembugamoorthy and Chandrasekaran 1986].

More specifically, AUTOGNOSTIC is inspired to a large extent by the model-based process of Kritik and Kritik2 for design adaptation [Bhatta and Goel 1994, Goel 1989, Goel 1991a, Goel 1991b, Stroulia and Goel 1992a, Stroulia and Goel 1992b]. Kritik, Kritik2 and AUTOGNOSTIC adapt devices, physical in the cases of Kritik and Kritik2 and abstract in the case of AUTOGNOSTIC, the functioning of which they "understand" in terms of qualitative SBF models.

The SBF models of Kritik and Kritik2 are based on an ontology of components, substances ontology and describe the internal causal processes of the device which result in the accomplishment of its external behaviors. These internal causal processes, in turn, arise from the transformation of the device substances by the device components. The SBF language of AUTOGNOSTIC is based on an ontology of tasks, methods and knowledge and describes the internal reasoning processes of the device which result in the accomplishment of its external behaviors. In problem solvers, the internal reasoning behaviors arise from the transformation of information by the problem-solver's tasks as it flows through the task structure. The causal behaviors of physical devices are, in general, deterministic, where in multi-strategy problem solvers the reasoning behaviors are non-deterministic.

The blame-assignment process in AUTOGNOSTIC's reflection is similar to the blame-assignment process of Kritik2, in that they both rely on the feedback, that is the output behavior desired by the artifact which is being redesigned, and the model of this artifact to identify the elements that are potentially at fault. However, AUTOGNOSTIC's blame- assignment process admits the possibility that the feedback may necessitate the acquisition of new domain knowledge where Kritik2 assumes that all the elements in the desired function are known. Furthermore, AUTOGNOSTIC's blame assignment considers as a potential cause of the failure the erroneous selection among the methods available for the failed task, where Kritik2 does not since there is a single behavior responsible for a given function.

Finally, the redesign process in AUTOGNOSTIC's reflection is quite different from the repair process in Kritik or Kritik2. The repair process in Kritik and Kritik2 uses a set of plans which can substitute a component or a substance in the failing device with another one, or introduce a sequence of similar components in order to extend the range of the device functioning. These plans however, do not modify the nature of the components' operation and do not modify the number and type of substances that flow through them. IDeAL [Bhatta 1995] builds on Kritik and Kritik2 and is able of such more drastic modifications to the device functioning (for a more extensive discussion on IDeAL see related paragraph in section 10.2). AUTOGNOSTIC's redesign process can introduce new subtasks in the problem-solver's task structure or introduce new inputs to existing subtasks, or modify the very information-transformation that existing subtasks perform in the overall context of the problem-solver's reasoning.

It is interesting to note that Kritik's, and as a sequence AUTOGNOSTIC's, SBF models are descendants of the Functional Representation scheme developed by Sembugamoorthy and Chandrasekaran [Sembugamoorthy and Chandrasekaran 1986] which they integrate with the ontology developed in the context of the Consolidation Theory [Bylander and Chandrasekaran 1985]. Functional Representation provided the main organizational primitives for organizing the designer's comprehension of a device functioning. Namely, the behavior of the device is a causal sequence of states which explain how the structure of the device achieves the function of the device. Each transition can be accomplished by a function of a sub-device, or by a more detailed behavior. The function is the intended purpose of the device, and provides an encapsulation mechanism for these behaviors. The structure is the set of its components and their connectivity. In the functional representation scheme, the functions act as indices for the behaviors that produce them. On the other hand, the consolidation theory [Bylander and Chandrasekaran 1985] described of a set of conceptual primitives which can be used to describe behavior, and a set of causal patterns which can be used to compose them.

Another descendant of the Functional Representation scheme can be found in the work of Allemang [1990] . In a spirit similar to AUTOGNOSTIC's theory, he has viewed computer programs as devices, and, has modeled such programs in terms of Chandrasekaran's Functional Representation scheme. The overall task of Dudu (Allemang's system) is very similar to AUTOGNOSTIC's. They both model computer programs, although very different aspects of them. Dudu's models specify the functional and compositional semantics of the programs at the level of programming-languages constructs, where AUTOGNOSTIC specifies the same semantics at the level of information processing and is agnostic of the implementation of the processing elements in the actual software system.

CHAPTER XI

CONCLUSIONS

The capability of learning is a prerequisite for autonomy. Rarely do autonomous intelligent agents, natural or artificial, who live and solve problems in realistic environments have perfect knowledge (i.e., complete, correct, and appropriately represented and organized knowledge) of their world, and perfect problem-solving architectures (i.e., complete, correct, and appropriately organized set of functional elements) for using this knowledge. The deficiencies in domain knowledge and problem-solving architecture often lead to performance failures, which constitute opportunities for learning. Upon failure, an intelligent agent needs to recover from the failure, and has the opportunity to learn and improve its knowledge and problem solving, so that it does not fail under similar conditions in the future.

This thesis demonstrates that reflection plays a key role in failure-driven learning. More specifically, it shows that, if an agent has a model that captures the functional and compositional semantics of its problem solving, then, when it fails, it can appropriately redesign itself and thus improve its performance.

11.1 Results

In Chapter 1 we had analyzed the problem of failure-driven learning in terms of the following dimensions:

problem-solving solving during to the agent to modifications, failure to have overall

1. the complexity of the problem solvers to which the learning process is applicable,
2. the dimensions along which the problem-solving performance can be improved,
3. the stages of the problem solving during which failure can be detected,
4. the types of feedback that the learning process may require,
5. the learning tasks the learning process is able to address,
6. the different types of adaptation strategies that the learner may employ in accomplishing its tasks, and
7. the extent to which the learning process prevents any undesired side affects of the adaptation strategies from occurring.

Let us now revisit these issues and discuss the answers that this thesis provides to them.

11.1.1 Complexity of Problem Solving

In general, the complexity of failure-driven learning increases as the complexity of the problem-solving process, whose performance it aims to improve, increases. Different problem-solving strategies suffer from different types of errors and necessitate different types of adaptations to the problem solver. In order to be able to effectively adapt a failing problem solver, a learning strategy

must be in some way aware of this problem-solver's reasoning strategy. One way, that a learning strategy can be made aware of the problem-solving strategy it is expected to adapt, is implicitly, by making assumptions regarding the particular steps of this problem-solving process and the types of errors that can possibly occur in it. But any learning strategy that relies on such implicit knowledge about problem solving is limited to the particular problem-solving strategy that meets the assumptions made. Thus, it becomes insufficient in the case of multi-strategy problem solving where a variety of problem-solving strategies may be used, that may not necessarily conform with the assumptions implicitly embedded in the learning strategy.

An alternative to making implicit assumptions about the problem-solving process is to analyze that process and explicitly represent explicitly all the knowledge relevant to it so that the learning process may use it. The latter alternative has the major advantage that the learning strategy is able to adapt a well-defined class of problem-solving strategies, namely the ones explicitly represented. In comparison, learning strategies of the former type are particular to a specific problem-solving strategy and fail unpredictably when the problem solver employs a strategy that does not meet their assumptions. The generality of the class of problem-solving strategies that can benefit from such a learning strategy, and consequently the generality of the learning strategy itself, depends on the expressiveness of the theory used to analyze problem solving.

The reflective learning process developed in this thesis exemplifies this later approach. It does not make any assumptions regarding the strategies that the problem solver may employ, or their number, or the tasks the problem solver may accomplish, or the domain in which it may live. It requires, however, that the problem-solving process should be analyzed in terms of the task-structure theory of problem solving, and should be modeled in its SBF language. Thus, the expressiveness of the SBF language defines the class of problem solvers that can benefit from this learning process. This thesis demonstrates that the SBF language is expressive enough to describe a wide class of problem solvers, across the spectrum from deliberative to reactive problem solving. The SBF language enables the specification of any problem solver, as long as, all the design elements of its functional architecture can be identified, and their individual functionalities and their overall composition can be specified. A very important aspect of the reflective learning process of this thesis is the way it treats the deliberative-reactive distinction. Traditionally, deliberative problem-solving behaviors have been seen as fundamentally different from reactive behaviors. Consequently, the learning methods developed for these two classes of problem solving have been completely different. The device stance to problem solving provides a unified view to both types of problem-solving behavior, and the design stance to learning gives rise to a learning method that can benefit both of them.

For an extended discussion on the example problem solvers that were analyzed in terms of the SBF language and used as testbeds for evaluating AUTOGNOSTIC, the system reifying the reflective learning process of this thesis, and the process of modeling a problem solver in that language, the reader can see Chapter 4.

11.1.2 Dimensions of Performance Improvement

In principle, there are three dimensions along which the performance of a problem solver may need to be improved: efficiency of its problem-solving process, quality of the solutions it produces, and size of the population of problems it can solve. Consider for example, the problem of inefficiencies in the problem-solving process. Quite often the reason for such inefficiencies is that the different steps of the process are not correctly organized and the information produced by one step is not effectively communicated to the other. In such cases, the process steps usually have to be reorganized. On the other hand, when the problem solver does not produce the right kind of solutions, or when it does not solve some of the problems it is presented with, then the very functionalities of some of its process steps may have to be modified. In the case of solution quality improvement, this is because to produce better solutions, some of the process steps may have to specifically search for the desired kind of solutions. In the case of extension of the class of solvable problems, some of the process steps may need to produce additional information necessary for solving the extended class of problems. Thus, in general, different dimensions of performance improvement necessitate the accomplishment of different types of learning tasks. In turn, these tasks give rise to different knowledge needs.

The ability to appropriately reorganize the problem-solving steps requires that the agent knows how these steps are organized in the first place. Even more importantly, it necessitates that the agent should know what the possible meaningful organizations are, and how they affect the problem-solving process. Thus, when these steps are reorganized, the modified organization is "legal" and an improvement over the former one. The agent's ability to modify the functionality of its own problem-solving steps, so that they produce "better" or more information, requires knowledge regarding the current functionality of these steps and the way in which they are combined to produce the overall.

The SBF language developed in this thesis is an example of a language designed to capture the above types of knowledge. The functionality of the problem-solving steps is captured in terms of the input and output information of a task and the functional semantics of the transformation between the two. Their current organization is captured in terms of the methods used to decompose these tasks. Their possible acceptable organizations are captured in terms of alternative methods with valid information and control interactions among their subtasks. The reflective learning process of this thesis relies on the functional and compositional semantics of the problem solver to adapt the problem solver and improve its performance along all three dimensions.

In the case of inefficient problem solving, the knowledge of the problem-solver's functional semantics enables the learning process to recognize when a particular step does not deliver the information expected of it. Furthermore, the knowledge of the problem-solver's compositional semantics enables it to relocate the failing step in such a point in the overall problem-solving process that it is invoked only in situations where it can actually deliver its expected functionality.

In the case of unsatisfactory solution quality, the knowledge of the problem-solver's functional semantics enables the learning process to recognize when a particular step does not produce the types of information that would enable the overall problem-solving process to produce the desired solution, although it delivers the functionality intended of it. This recognition, in turn, enables it to postulate a new functionality for the failing step. The problem-solver's compositional semantics enable it to effectively integrate the desired new functionality in the current process. With its functionalities enhanced, the problem solver is tuned to producing the kind of solutions desired of it and meet goals it was not explicitly designed for.

For an example of how this knowledge is used to improve the problem-solver's efficiency, the reader may go back to example problem 3, discussed in detail in sections 5.2.3, 6.4, and 7.5.1, and also revisited in 9.5. For an example of how is is used to improve the quality of the problem-solver's solutions, the reader may go back to example problem 1, discussed in detail in sections 5.2.1, 6.6.2, and 7.5.4. For a detailed discussion on the experiments conducted with AUTOGNOSTIC and the improvements that AUTOGNOSTIC brought about the problem solvers it was integrated with, the reader may see Chapter 9.

Finally, it is important to note that the effectiveness of the reflective learning process with respect to these three types of performance improvement (see section 9.3) relies on the fact that it admits the possibility that the behavior desired of the failing problem solver is not already within its competence. In comparison, most trace-based methods assume that the problem solver is able to exhibit the desired behavior, and thus preclude the agent from extending the class of problems it can solve.

11.1.3 Failure Detection

The failures that can trigger learning are of two kinds: failures that the agent itself is able to recognize during its problem solving, and failures that are brought to the agent's attention by its external environment at the end of its problem solving. An agent's ability to recognize at least some of its failures independently, i.e., without external probing, increases the agent's autonomy. Furthermore, it also extends the situations in which the agent can learn, since the agent does not need a "supervisor" to notice its failures. To be able to independently recognize its own failures, an agent must have some knowledge to distinguish between a successful and a failed problem-solving episode. One way to do that, is by having some a-priori knowledge regarding the acceptable final states that the agent may reach for each problem that it solves. This knowledge however can only be used at the end of the problem solving, and the agent may flounder for a long time before reaching

the point where it can use it.

Alternatively, the agent may have an abstract specification of what constitutes a correct problem-solving process, or more specifically, what constitutes a correct problem-solving step. Such knowledge would enable the agent to compare and evaluate the actual outcomes of each step in the context of a particular problem-solving episode against its specification. Thus, in addition to recognizing errors in its problem solving, the agent would also be able to evaluate its progress while reasoning.

The functional semantics of the problem-solver's tasks, as specified in the SBF language, constitute exactly that type of knowledge. That is, they specify at an abstract, problem-independent level, what constitutes a correct problem-solving process. The reflective learning theory of this thesis demonstrates how this specification can be used to monitor and evaluate the problem-solving progress. The functional semantics of the problem-solver's subtasks enable the evaluation of the intermediate results of its problem-solving process, i.e., the intermediate types of information. In addition, because the SBF language enables the specification of the problem-solving process at several different levels of abstraction, the learning process can evaluate each piece of produced information against the functional semantics of a set of recursively nested tasks to which this information is relevant. Thus, the reflective learning process can identify failures at different levels of the problem-solving task hierarchy.

For an extended discussion on the monitoring process and the use of functional and compositional semantics in evaluating the progress of problem solving, the reader may refer to Chapter 5 and section 8.1.

solving and

11.1.4 Types of Feedback

In general, an agent may require several different types of feedback in order to learn from its failures. Different types of feedback differ with each other with respect to their information content. The least descriptive type of feedback is a simple success/failure message. In increasing order of information content, the feedback can include the preferred solution, or the preferred solution and a pointer to where in the problem-solving process the error might have occurred, or finally, a complete trace of the problem-solving process as it should have been correct. The higher the information content of the feedback a learning process requires, the narrower the knowledge conditions it is applicable.

The reflective learning process of this thesis does not make any assumptions beyond the preferred solution. In fact, it provides a partial account on how an agent might identify errors in its own process and repair itself without any external feedback. This learning theory can address failures with or without feedback from the external environment, although to a different degree. When it does receive feedback from the environment, it requires it in the form of the solution desired of the problem solver for the particular problem. The learning process does not assume that the desired solution is of any particular quality. In fact, it can receive feedback indicative either of a quality-of-solution criterion, or of a process-efficiency criterion. Finally, it is important to note that, in some cases this reflective learning process is able to produce its own feedback. This ability, relies on its knowledge of the domain in which the desired solution belongs, as specified in the SBF model of the problem solver. The SBF model specifies the domain of values that the different types of information produced during problem solving may take. Based on this knowledge and the characteristics of the solution, as specified by the functional semantics of its producing tasks, the learning process is sometimes able to postulate potentially desired solutions instead of the failed one.

For an analysis of the capabilities of the learning process with and without feedback, the reader may refer to Chapter 6. For the types of criteria that the feedback solution may indicate, the reader should see section 9.4.2. For a discussion on how the reflective learning process may produce its own feedback, the reader may look in section 6.4.

11.1.5 The Learning Tasks

An issue of critical importance to the effectiveness of the learning process is the degree to which this process can effectively identify the errors that cause the problem-solver's failures, and autonomously spawn learning tasks appropriate for eliminating them. Note that, such errors may lie anywhere in the agent's domain knowledge or even in its functional architecture. Thus, an effective learning process should be able to perform learning tasks that can address instances of both types of errors. Often, learning theories make assumptions regarding the types of the problem-solver's domain knowledge that may be faulty and focus their attention in repairing these types of knowledge only. With respect to errors in the problem- solver's functional architecture, most learning theories assume that the problem solver has a complete set of functional elements, although they may admit the need for their reorganization.

The ability to recognize errors in all the types of the problem-solver's domain knowledge requires that the agent knows about the different types of knowledge used in problem solving. In addition, it requires knowledge regarding the roles that each type plays in problem solving so that different failures may potentially be traced back to errors in different types of knowledge. The ability to reason about and modify the functionalities of the elements of its own functional architecture requires that the agent should know about the functionalities that these elements were originally designed with. In addition, the agent should also comprehend the way in which these elements are composed to deliver the output desired of the whole problem-solving process, so that it can assign blame for the deficiencies of the overall process to deficiencies of its elements.

The reflective learning theory of this is able to accomplish a taxonomy of learning tasks that cover indeed both these types of adaptations. It includes learning tasks that modify and improve the content and the organization of the problem- solver's domain knowledge, as well as the content and the organization of its functional architecture. This taxonomy is grounded on the SBF language for analyzing and specifying problem solving, in two ways. First, the language used to explicitly represent the learning tasks, potentially relevant to a failing problem solver, arises from the SBF language. Second, the knowledge necessary for localizing the potential cause of the failure and postulating the need for a specific learning task is captured in the SBF model of the problem solver.

The blame-assignment method of the reflective learning process relies on the SBF model of the problem solver. When the problem solver fails to produce the solution desired of it and is given as feedback the preferred solution, the functional semantics of the overall task enable the blame-assignment method to evaluate whether or not the desired solution is within the class of solutions the problem solver is designed to produce. Based on the compositional semantics of the problem-solver's task structure, the blame-assignment method is able to incrementally focus its investigation at lower, more detailed levels of abstraction. Thus, it inspects an increasing amount of information interactions among the problem-solver's subtasks on an as-needed basis. The functional semantics of the lower-level subtasks enable it to identify which subtasks were correctly performed. Thus, it can acquit large segments of the problem-solver's process, and move the focus of its investigation into earlier stages of the process. This ability to acquit large parts of the problem-solving process, and the parts of the task structure responsible for producing it, addresses also the issue of the efficiency of the blame-assignment method, which is especially important given the potential complexity of the problem solvers that it deals with. Finally, the functional semantics of the intermediate subtasks of the problem solver enable the blame-assignment method to infer alternative values for the intermediate types of information that could enable the problem solver to derive the overall desired solution. Thus, the blame-assignment process is able to localize the global requirements of the environment for an overall desired solution to local requirements for a specific desired intermediate information.

The blame-assignment method of this reflective learning process is able to identify potentially several causes for the problem-solver's failure, and autonomously spawn an equivalent number of learning tasks. The types of causes that this method is able to identify, and the knowledge conditions under which it does so are summarized below:

Incomplete or Incorrect Domain Knowledge: The feedback that the learning process receives may convey information regarding the current state of its domain, which is missing from the agent's

domain knowledge. Based on its comprehension of the types of objects that exist in its domain and the types of relations that are applicable to them, the blame-assignment method is able to recognize gaps in the content of this knowledge.

Erroneous Organization of the Domain Knowledge: As the agent assigns blame for its failure to produce the desired solution based on its comprehension of the functional semantics of its tasks, it is able to derive the inferences that could have potentially led to the desired solution. When these inferences rely on the organization of its knowledge, that is when the functional semantics of the task responsible for the inference are expressed in terms of some domain-organization relation, the blame-assignment process postulates that the reason they were not drawn may be because this organization is imperfect.

Erroneous Organization of the Task Structure: Sometimes the problem solver fails to successfully complete its reasoning on a particular problem, although its problem decomposition is seemingly correct with respect to the compositional semantics of its task structure. In such cases, the blame-assignment method postulates that the organization of the problem-solver's task structure may be incorrect, in that it may ignore critical information interactions among these tasks.

Under- or Over-Constrained Task Functionality: Finally, based on the functional and compositional semantics of the problem-solving task structure, the blame-assignment method is able to infer that some particular task should be able to produce a specific value for an intermediate type of information, in order for the preferred solution to be produced. When the value of this type of information belongs in the class of values that the failed task is designed to produce, the blame-assignment method postulates that the cause of the failure may be that the functionality of this task is insufficiently specified. Such an under-specified task might ignore the correct inference in favor or other possible inferences. Alternatively, when the value of the information is outside the class of values expected of the failed task, the blame-assignment method postulates that the cause of the failure may be that the functionality of this task is overly specified. Such an over-specification could potentially result in the non-production of the complete class of values desired of the task.

The last type of learning tasks is especially difficult to spawn. In order for the agent to recognize the need to modify the functionality of an element in its functional architecture, it has to be able to establish that the inferences produced by this element are not correct with respect to some requirements imposed on the overall problem-solving process. This requires that the agent understands the nature of the inferences that each task is designed to contribute in the overall problem-solving process. In this reflective learning process, this knowledge is captured by the functional and compositional semantics of the problem-solving task structure. This knowledge enables it to localize the failure of the problem solver to exhibit the behavior desired of it to the failure of some subtask to draw the inferences necessary for that type of behavior.

For a detailed discussion on the role of functional and compositional semantics in the blame-assignment process, the reader may look in section 6.6. For a discussion on how the SBF language for modeling problem solving gives rise to the taxonomy of learning tasks addressed in this thesis, the reader should refer to section 3.2.

11.1.6 Learning Strategy and Methods

When a learning theory admits that there can potentially be several causes for each particular failure in the problem solving, two very important issues arise. First, the learning theory has to account for how one (or potentially more) of these causes is selected to be eliminate first. Essentially, given a set of possible options on adaptations that could potentially eliminate the failure, the learning process should be able to decide what to learn. Second, if these causes are of different types, which is quite often the case in multi- strategy problem solvers, then a single learning strategy is likely to be insufficient for addressing all of them. Thus, there is a need for multiple learning strategies. Then, having decided what to learn, the learning process can decide on how to learn. These two issues necessitate that the learning theory should be general enough to accommodate the use of

multiple learning strategies and that it should account for a type of knowledge on the basis of which the learner can select what to learn and how, in the context of a particular failed problem-solving episode.

The reflective learning theory developed in this thesis is an example of such a theory. It integrates seamlessly the use of multiple learning strategies, each one appropriate for a different set of learning tasks. In addition, it proposes a set of criteria for selecting which among the potential causes of the problem-solver's failure to eliminate first, and an additional set of criteria for selecting which strategy should be used to eliminate it.

The criteria for selecting which learning task to address first arise from the task-structure theory of problem solving. The task-structure theory gives rise to an ordering of the "gravity" of the possible causes of the failure. For example, errors in the content of the agent's domain knowledge are the most critical. This is because, as long as the problem solver does not have the knowledge relevant to producing the desired solution, it will fail, irrespective of whether or not the rest of its knowledge is organized correctly and whether or not its task structure is correct. Based on this ordering, the learning process selects to address the most difficult potential cause, with the rationale that, if this is indeed the cause of the failure, then eliminating any other secondary cause will not correct the failing behavior of the problem solver.

The next issue, that is, which learning strategy to invoke for eliminating the selected cause, is also resolved, at least in part, based on the task-structure theory of problem solving. The task-structure theory gives rise to an ordering of the "cost" of the possible adaptations that can be performed on the problem solver. This ordering depends mainly on the amount of information each adaptation requires, and on the extent to which each individual adaptation gives rise to new ones. Thus, when the selected cause can be potentially eliminated by a variety of different strategies, the learning process selects the least expensive strategy whose knowledge requirements are met in the particular context.

The learning strategies integrated in this reflective learning process and the knowledge that makes them possible are summarized below:

Plan for knowledge acquisition of a new domain object: This plan is applicable when the cause of the failure is the fact that the agent does not know about a particular domain concept, necessary for the production of the preferred solution. The meta-level knowledge regarding the problem-solver's domain captured in its SBF model, that is, the types of domain concepts, the relations that pertain to each type, and the constraints that must hold true among these relations, enables this plan to consistently integrate the new object in the problem-solver's knowledge base, by updating all the relations that pertain to its class, and by propagating the constraints that apply to them.

Plan for updating a domain relation: This plan is invoked when the cause of the problem-solver's failure is the incorrectness or the incompleteness of its knowledge regarding a domain relation. This relation could refer to the state of the world, in which case the cause is really incomplete or incorrect domain knowledge, or it could be an indexing relation, in which case the cause is really incorrect knowledge organization. Again, the meta-level knowledge regarding the problem-solver's domain enables this plan to consistently modify the agent's knowledge of associations between objects in its domain, so that it can draw the inferences desired of it.

Plan for task-structure reorganization: This plan modifies the relative organization of the tasks in the agent's task structure, in order to make explicit information and control interactions between them that are currently hidden. The compositional semantics of the problem-solver's task structure enables it to reorganize the task structure while also maintaining its overall consistency.

Plan for task substitution: This plan is applicable when the cause of the failure is the under- or over-constrained functionality of a problem-solving task. To substitute a failing task, the learner uses its knowledge of "families of tasks" to identify a member in the family of the failed task that could potentially deliver the desired behavior in the context of the failed problem-solving episode.

This knowledge is captured in terms of prototype tasks and their instances in the SBF model of the problem solver.

Plan for task modification: This plan is applicable when the cause of the failure is the under- or over-constrained functionality of a problem-solving task. The learning process uses the compositional semantics of the problem-solver's task structure to identify the types of information available to the problem solver at the point when it attempts to accomplish the failed task. In addition, it uses its meta-level knowledge regarding the problem-solver's domain to postulate what the semantics of the failing task should be. When it is able to infer such semantics, it modifies the current set of functional semantics of the failed task by replacing the failing relation, or by including a new relation in the set.

Plan for selection-task insertion: This plan is applicable when the cause of the failure is the under-constrained the functionality delivered by a task of the problem solver. Like the plan above, it relies on the agent's ability to formulate a specification of the functionality desired of the failed task, based on its meta-model of the domain. In this plan however, a new task is introduced in the task structure whose role is to refine the class of inferences desired by the currently failing task. As with the task-structure reorganization plan, it is the compositional semantics of the task structure that enables the learning process to modify it (by introducing a new element in it) while not compromising the overall consistency of the problem solver.

This reflective learning theory seamlessly integrates a wide range of learning tasks, some of which have been already addressed in isolation by earlier AI research. By investigating all of them together, this learning theory provides a more constrained account of when each one of them is relevant, and of how they interact with each other.

For a more extended discussion on how the learning process selects what to learn, the reader should refer to section 7.1. For a more extended discussion on the issue of how a particular learning strategy is chosen over an alternative one, the reader should refer to section 7.2.

11.1.7 Maintaining the Consistency of Problem Solving

Finally, a very important issue in adapting a failing problem solver is the issue of ensuring that the adaptations performed to it do not compromise the consistency of its problem solving. This problem is especially difficult when the set of functional of the problem solver must be modified, as in the case of introduction of new tasks.

The ability to reason about the potential consequences of such an adaptation and perform it in a way that does not compromise the overall consistency of the problem-solving process, necessitates that the agent has an understanding of the elements of the problem-solver's functional architecture. The learning process should be aware of the functional role of these elements and the way in which their functioning gets composed in the overall problem-solving process. Thus, when a new element is to be introduced, it can decide first, what its functional role should be by evaluating what type of functionality is missing from the overall problem-solving process, and second how this new element should interact with the existing ones.

The SBF model of the problem solver captures such knowledge regarding the problem-solver's elements. For each task of the problem solver, it specifies the types of information it requires as input, the types of information it contributes as output, and its functional semantics, i.e., the type of the transformation that the task performs to produce the latter from the former. The task specifications capture the information-flow interactions among the problem-solver's tasks. In addition, in the context of the problem-solver's methods, the model specifies the control interactions among them, i.e., their compositional semantics. The learning uses this knowledge to reason about the information-flow and control interactions of the new element with the rest of the problem solver. The locality of the problem-solving process where the new task should be introduced is decided by the blame assignment which identifies the locality of an existing malfunctioning task. The blame assignment also identifies the type of information which is incorrectly produced in the failing problem solver, which will constitute the output of the new element. To identify the functionality

desired of the new element, the learning process consults the SBF model to identify what types of information are available in that locality. Based on the problem-solver's compositional semantics, it identifies the tasks that precede the failing one and from their SBF specification, it collects the types of information they produce. These types of information could potentially constitute the new element's input. From these types of information, and from its desired output the learning process postulates a specification for the type of its information transformation. At this point, the information-flow interactions of the new element with the rest of the problem solver have been specified. Then, based on the specification of the method that spawns the faulty task, the learning strategy decides the details of the control interactions of the new element with the existing ones, in a manner that the overall composition of the problem-solver's functional architecture is correct. For an involved discussion on the role of the SBF model in maintaining the consistency of the problem solver and its SBF model the reader should refer to Chapter 7.

identify not is under-

11.2 Critique and Future Directions

11.2.1 Immediate Future Research

Section 9.8 of Chapter 9 discusses the limitations of AUTOGNOSTIC in detail. Some of these limitations pertain to AUTOGNOSTIC as a system while others pertain to the theory embodied by AUTOGNOSTIC. Addressing these limitations formulates the immediate research agenda.

The most immediate item in this agenda is the extension of the expressiveness of the language for task semantics, to allow the specification of semantic relations among multiple types of input and output information. The current language is limited to tasks in which the contribution of each input to the task output can be linearly separated. This is a limitation of AUTOGNOSTIC as a system, and the different steps necessary for extending the system are sketched out in fair amount of detail in section 9.8. This extension will necessitate extensions to the blame-assignment method, and also to the method for discovering new semantics. Once this extension has been completed, it would be of interest to compare the current version of AUTOGNOSTIC with the new, extended one, in order to identify the additional coverage that this extension might provide to the reflection process, and the additional complexity that it might incur to the learning process.

The next issue is the problem of modifying the input-output relations of the high-level tasks of the problem solver. The modification of the overall task input appears to be within the current capabilities of AUTOGNOSTIC. However, to date there have been no problem-solving scenarios where this capability has been put to test. The first step would be to devise a scenario where this type of adaptation would be useful, and investigate AUTOGNOSTIC's behavior. The next step would be to extend the learning process so that it can integrate new types of inferences in the problem-solver's task structure such that they produce additional information than the current output of the overall task. To this end, several approaches seem promising given the current status of the learning method. First, the notion of different task instantiations of prototypical tasks could be extended to include tasks which share some common inputs and outputs but each task instance could also have additional inputs or outputs. Consequently, the plan for task substitution could be employed to replace a task with a instance of the same prototypical task which delivers the desired additional outputs. An alternative approach would be to extend the learning task to introduce new types of output information in the task structure, when the problem solver is presented with problems which are extensions of the problems it is already solving. It could then use its process for postulating semantic relations, to discover how this new type of information relates with the rest of the information context.

Finally, an interesting problem, close to the theory currently implemented in AUTOGNOSTIC, is the re-representation of the domain knowledge, or alternatively the problem of modifying the ontology of the problem solver. This is a learning task which arises naturally in the AUTOGNOSTIC's SBF language. There are two interrelated issues that must be addressed by any learning method which aspires to provide a solution to this problem: first how does the system even recognize the need for such a change, and second, how does it incrementally modify its representation

scheme without radically disturbing the knowledge represented in its current ontology. At this point AUTOGNOSTIC provides a partial answer to the first question: the need for re-representation of the domain knowledge becomes evident from the failure of the system to parse some information provided by its external environment.

11.2.2 Extended Future Research

Knowledge-Based Systems: Knowledge-based systems are one class of intelligent systems which solve complex problems and which are often reused in environments slightly different than the ones in the context of which they were developed. Such reuse is not completely without problems, however it does not usually require complete redesign of the system. Even in the original environment in which it was designed, given the passing of a long period of time, a knowledge-based system may start to fail due to changes in its environmental conditions. Thus, the problem of maintaining knowledge-based systems over a long time span, and re-situating them in new contexts becomes a natural candidate for the reflection process developed in this thesis.

The most important challenge that real knowledge-based systems will present to the reflection process of this thesis is scalability. The problem solvers AUTOGNOSTIC was integrated with are quite sophisticated, but do not have the amount of knowledge that systems deployed in the real world have. If the re-situation of the system fails mainly due to errors in the problem-solving task structure, then the abundance of domain knowledge may be an advantage since it may give rise to a lot of failures stemming of the same cause, and thus a more constrained modification of the task structure may be performed. If, on the other hand, the cause of the failure lies mainly in the knowledge of the system, then the cost of reflection might be excessive. A potential approach, applicable in cases where the major cause of failure is known, would be to instantiate different versions of the reflection process, ablated along different dimensions.

Robotics: Autonomous agents are yet another candidate for evaluating the effectiveness of the reflective learning process as a self-adaptation mechanism. The REFLECS experiment was a first attempt in that direction, and it gave rise to some interesting issues particular to autonomous agents.

First, how does reflection and action interleave? In disembodied agents who reason based on internal representations of the world but do not act in it, this question is not even meaningful. However, in the context of robotics where the resources available to reflection are limited by the need to act timely, this interleaving becomes critical.

Second, how does noise in the perceptual input affect the reflection process? The quality and the effectiveness of the adaptations that the learning process performs to the problem solver depends on the feedback that this process receives. If the channel that perceives the feedback is noisy, then the learning process might misinterpret the inference desired of it, and it might even adapt the functional architecture towards the wrong direction. One potential approach to this problem might be to extent the learning process to verify its feedback before proceeding to use it as the basis of the agent's adaptation.

Finally, in the real world time often becomes of critical importance. AUTOGNOSTIC's SBF language does not capture any notion of time, and therefore, the learning method cannot recognize errors that are the result of the interactions among tasks in the time line. In the context of autonomous robotics, the SBF language may have to be extended to explicitly represent time-overlap between tasks, delays, deadlines and similar concepts.

Software Engineering: Artificial problem solvers are computer programs. Consequently, modeling the behavior of a problem solver in terms of a SBF model requires the acquisition of SBF models for the source code implementing the problem solver. Further, modifying the SBF model of a problem-solver's task structure implies modification of that source code. To date, the implementation of the reflection process in AUTOGNOSTIC, assumes an already developed model for the system, and, sometimes, it also assumes the cooperation of a human programmer to perform parts of the suggested modifications to the system's code. However, in principle, program-understanding

techniques could enable the initial boot-strapping of the process by providing methods for acquisition of SBF models from code, and automatic-programming techniques could help to make the learning process more autonomous.

There are two particularly interesting lines of research, in program-comprehension and program-debugging correspondingly, which could naturally be integrated with the reflection process of this thesis. Wills [1992] has developed a program-understanding method based on the recognition of programming "cliches" in the source code of a software system. These "cliches" are syntactic patterns, and consequently the "understanding" of the program that this method is able to produce lacks explanatory power. In subsequent research, Wills is making efforts to include functional semantics to the description of "cliches" which would make them quite similar to tasks. In addition, Allemang [1990] has developed a program-debugging method based of a Functional Representation of the program under modification. SBF models are descendants of the original Functional Representation scheme [Sembugamoorthy and Chandrasekaran 1986], and therefore the organization of Allemang's models is very similar to AUTOGNOSTIC's SBF models, with the exception that their semantics are based on the operational semantics of the programming language in which the program is implemented. The common principles along which these models are organized could make possible the integration of the two types of models into a single one, analyzing the problem solver both at the knowledge and at the implementation levels.

Design: Finally, the future-research agenda includes the task of closing the circle from design to learning and back to design. The reflective learning process developed in this thesis is essentially a model-based, redesign process. As such, it must have implications to design in general.

The reflective learning process of this thesis raised several interesting issues which are not clear whether or not they have a counterpart in the domain physical devices. Problem solvers can use multiple strategies to solve their problems. What is the counterpart of a multi-strategy reasoner in the domain of physical devices? In fact, the notion of such a counterpart seems unnatural. What are the implications of the lack of such a counterpart to the cost of the blame-assignment and repair processes? The reflective learning method is able to discover new tasks that should be integrated in the problem solver's task structure. How can this process translate to discovering new components that do not already exist to introduce new functionalities in the internal causal processes of the device?

On a different line of thought, the integration of AUTOGNOSTIC with KRITIK2 and AUTOGNOSTICONKRITIK2's ability to reflect upon its own failures, learn from them, and improve its performance raises some interesting questions regarding the potential usefulness of reflection based on SBF models on design education, which the Canah-Chab project has just started to explore.

APPENDIX A

THE SBF MODEL OF ROUTER'S PATH PLANNING

TASKS

name: ROUTE-PLANNING
methods: (ROUTE-PLANNING-METHOD TRIVIAL-ROUTE-PLANNING-METHOD)
instances: (PATCH-FRONT PATCH-END)
input: (INITIAL-POINT FINAL-POINT TOP-NEIGHBORHOOD)
output: (DESIRED-PATH)
semantics: (SAME-POINT INITIAL-NODE(DESIRED-PATH) INITIAL-POINT)
(SAME-POINT FINAL-NODE(DESIRED-PATH) FINAL-POINT)

name: CLASSIFICATION
procedure: CLASSIFICATION-PROC
input: (INITIAL-POINT FINAL-POINT TOP-NEIGHBORHOOD)
output: (INITIAL-ZONE FINAL-ZONE)
subtask of: ((ROUTE-PLANNING ROUTE-PLANNING-METHOD))
semantics: (ZONE-INTERSECTIONS INITIAL-ZONE INITIAL-POINT)
(ZONE-INTERSECTIONS FINAL-ZONE FINAL-POINT)

name: PATH-RETRIEVAL
methods: (PATH-RETRIEVAL-METHOD)
input: (INITIAL-POINT FINAL-POINT INITIAL-ZONE FINAL-ZONE)
output: (MIDDLE-PATH INT1 INT2)
subtask of: ((ROUTE-PLANNING ROUTE-PLANNING-METHOD))

name: STORE
procedure: STORE-CASE
input: (INITIAL-POINT FINAL-POINT INITIAL-ZONE FINAL-ZONE DESIRED-PATH)
subtask of: ((ROUTE-PLANNING ROUTE-PLANNING-METHOD))

name: TRIVIAL-ROUTE-PLANNING
procedure: ROUTE-PLANNING-METHOD-TRIVIAL
input: (INITIAL-POINT FINAL-POINT)
output: (DESIRED-PATH)
subtask of: ((ROUTE-PLANNING TRIVIAL-ROUTE-PLANNING-METHOD))

name: RETRIEVAL
procedure: RETRIEVE-CASE
prototype: RETRIEVAL-PROTOTYPE
input: (INITIAL-POINT FINAL-POINT INITIAL-ZONE FINAL-ZONE)
output: (MIDDLE-PATH)
subtask of: ((PATH-RETRIEVAL PATH-RETRIEVAL-METHOD))
semantics: (SAME-POINT INITIAL-NODE(MIDDLE-PATH) INITIAL-POINT)
(SAME-POINT FINAL-NODE(MIDDLE-PATH) FINAL-POINT)

```
name:    RETRIEVAL-PROTOTYPE
instances:     (RETRIEVAL-PRIME1     RETRIEVAL-PRIME2     RETRIEVAL)
input:    (INITIAL-POINT     FINAL-POINT     INITIAL-ZONE     FINAL-ZONE)
output:     (MIDDLE-PATH)

name:    FIND-DECOMPOSITION-INIS
procedure:     FIND-INIS-FROM-MIDDLE-PATH
input:    (MIDDLE-PATH)
output:     (INI1   INI2)
subtask   of:   ((PATH-RETRIEVAL     PATH-RETRIEVAL-METHOD))
semantics:     (SAME-POINT     INI1   INITIAL-NODE(MIDDLE-PATH))
       (SAME-POINT     INI2   FINAL-NODE(MIDDLE-PATH))
conditions:     (NOT   (NULL   MIDDLE-PATH))

name:    SEARCH
methods:     (MODEL-BASED-METHOD
       CASE-BASED-METHOD)
input:    (INITIAL-POINT     FINAL-POINT     INITIAL-ZONE     FINAL-ZONE
       MIDDLE-PATH     TOP-NEIGHBORHOOD)
output:     (DESIRED-PATH)
instances:     (SOURCE-SUBPROBLEM     DESTINATION-SUBPROBLEM)
subtask   of:   ((ROUTE-PLANNING     ROUTE-PLANNING-METHOD))

name:    INTRAZONAL-SEARCH
methods:     (INTRAZONAL-SEARCH-METHOD     INTRAZONAL-METHOD-TRIVIAL)
instances:     (MIDDLE-SUBPROBLEM)
input:    (INITIAL-POINT     FINAL-POINT     INITIAL-ZONE)
output:     (DESIRED-PATH)
subtask   of:   ((SEARCH   MODEL-BASED-METHOD))
semantics:     (FOR-ALL   n IN NODES(DESIRED-PATH):
 (INVERSE   ZONE-INTERSECTIONS)     n INITIAL-ZONE)
conditions:     ((ZONE-INTERSECTIONS     INITIAL-ZONE     FINAL-POINT))

name:    INTERZONAL-SEARCH
methods:     (INTERZONAL-SEARCH-METHOD)
input:    (INITIAL-POINT     FINAL-POINT     INITIAL-ZONE     FINAL-ZONE     TOP-NEIGHBORHOOD)
output:     (DESIRED-PATH)
subtask   of:   ((SEARCH   MODEL-BASED-METHOD))
conditions:     (((NOT   ZONE-INTERSECTIONS)     INITIAL-ZONE     FINAL-POINT))

name:    PATH-FRONT
prototype:     ROUTE-PLANNING
input:    (INITIAL-POINT     INI1   TOP-NEIGHBORHOOD)
output:     (FRONT-PATH)
subtask   of:   ((SEARCH   CASE-BASED-METHOD))

name:    PATH-END
prototype:     ROUTE-PLANNING
input:    (INI2   FINAL-POINT     TOP-NEIGHBORHOOD)
output:     (END-PATH)
subtask   of:   ((SEARCH   CASE-BASED-METHOD))

name:    CER-PATH-SYNTHESIS
prototype:     RECOMPOSITION
```

```
input:    (INT1  INT2  FRONT-PATH    MIDDLE-PATH    END-PATH)
output:    (DESIRED-PATH)
subtask  of:   ((SEARCH   CASE-BASED-METHOD))

name:    RECOMPOSITION
procedure:     PATH-RECOMPOSE
instances:     (MBR-PATH-SYNTHESIS      CBR-PATH-SYNTHESIS)
input:    (INT1  INT2  FRONT-PATH    MIDDLE-PATH    END-PATH)
output:    (DESIRED-PATH)
semantics:     (FIRST-SUBPATH     DESIRED-PATH     FRONT-PATH)
       (SECOND-SUBPATH     DESIRED-PATH     MIDDLE-PATH)
       (THIRD-SUBPATH     DESIRED-PATH     END-PATH)

name:    DECOMPOSITION
methods:     (INTERZONAL-DECOMPOSE-METHOD)
input:    (INITIAL-POINT     FINAL-POINT     INITIAL-ZONE     FINAL-ZONE
      MIDDLE-PATH    INT1  INT2  TOP-NEIGHBORHOOD)
output:    (MIDDLE-PATH     FRONT-PATH     END-PATH)
subtask  of:   ((SEARCH   INTERZONAL-SEARCH-METHOD))

name:    MBR-PATH-SYNTHESIS
prototype:     RECOMPOSITION
input:    (INT1  INT2  FRONT-PATH    MIDDLE-PATH    END-PATH)
output:    (DESIRED-PATH)
subtask  of:   ((SEARCH   INTERZONAL-SEARCH-METHOD))

name:    INIT-SEARCH
procedure:     INITIALIZE-SEARCH
input:    (INITIAL-POINT)
output:    (POSSIBLE-PATHS)
subtask  of:   ((INTRAZONAL-SEARCH       INTRAZONAL-SEARCH-METHOD))

name:    PATH-INCREASE
prototype:     STEP-IN-PATH-INCREASE
input:    (POSSIBLE-PATHS     FINAL-POINT     INITIAL-ZONE)
output:    (POSSIBLE-PATHS     DESIRED-PATH)
subtask  of:   ((INTRAZONAL-SEARCH       INTRAZONAL-SEARCH-METHOD))

name:    STEP-IN-PATH-INCREASE
methods:     (PATH-INCREASE-METHOD)
instances:     (PATH-INCREASE)
input:    (CURRENT-PATHS     DESTINATION     A-ZONE)
output:    (NEW-CURRENT-PATHS       THE-PATH)

name:    INTRAZONAL-TRIVIAL
procedure:     ROUTE-PLANNING-METHOD-TRIVIAL
input:    (INITIAL-POINT     FINAL-POINT)
output:    (DESIRED-PATH)
subtask  of:   ((INTRAZONAL-SEARCH       INTRAZONAL-METHOD-TRIVIAL))

name:    FIND-LEAST-COMMON-SUBZONE
procedure:     LCSUBZONE
input:    (INITIAL-ZONE     FINAL-ZONE)
output:    (LEAST-COMMON-SUBZONE)
subtask  of:   ((DECOMPOSITION       INTERZONAL-DECOMPOSE-METHOD))
```

```
name:    FIND-SOURCE-SUBZONE
procedure:     FIND-SUBZONE
input:    (INITIAL-ZONE    LEAST-COMMON-SUBZONE)
output:    (SOURCE-SUBZONE)
subtask  of:   ((DECOMPOSITION     INTERZONAL-DECOMPOSE-METHOD))
semantics:     ((INVERSE   CHILDREN-ZONES)     SOURCE-SUBZONE     LEAST-COMMON-SUBZONE)

name:    FIND-SOURCE-COMMON-INTS
prototype:     FIND-COMMON-INTS
input:    (INITIAL-POINT     SOURCE-SUBZONE     LEAST-COMMON-SUBZONE)
output:    (INT1)
subtask  of:   ((DECOMPOSITION     INTERZONAL-DECOMPOSE-METHOD))
conditions:     (NOT  (EQUAL  SOURCE-SUBZONE     LEAST-COMMON-SUBZONE))

name:    FIND-COMMON-INTS
procedure:     FIND-COMMON-INTS
instances:     (FIND-SOURCE-COMMON-INTS        FIND-DEST-COMMON-INTS)
input:    (ANCHOR-INT    SUBZONE   COMMON-SUBZONE)
output:    (COMMON-INT)
semantics:     ((INVERSE   ZONE-INTERSECTIONS)      COMMON-INT    COMMON-SUBZONE)
       ((INVERSE   ZONE-INTERSECTIONS)      COMMON-INT    SUBZONE)

name:    DEFAULT-SOURCE-COMMON-INT
prototype:     DEFAULT-COMMON-INT
input:    (INITIAL-POINT     SOURCE-SUBZONE     LEAST-COMMON-SUBZONE)
output:    (INT1)
subtask  of:   ((DECOMPOSITION     INTERZONAL-DECOMPOSE-METHOD))
conditions:     (EQUAL  SOURCE-SUBZONE     LEAST-COMMON-SUBZONE)

name:    DEFAULT-COMMON-INT
procedure:     DEFAULT-COMMON-INT-PROC
instances:     (DEFAULT-SOURCE-COMMON-INT        DEFAULT-DEST-COMMON-INT)
input:    (ANCHOR-INT    SUBZONE   COMMON-SUBZONE)
output:    (COMMON-INT)

name:    FIND-DEST-SUBZONE
procedure:     FIND-SUBZONE
input:    (FINAL-ZONE    LEAST-COMMON-SUBZONE)
output:    (DEST-SUBZONE)
subtask  of:   ((DECOMPOSITION     INTERZONAL-DECOMPOSE-METHOD))
semantics:     ((INVERSE   CHILDREN-ZONES)     DEST-SUBZONE     LEAST-COMMON-SUBZONE)

name:    FIND-DEST-COMMON-INTS
prototype:     FIND-COMMON-INTS
input:    (FINAL-POINT     DEST-SUBZONE     LEAST-COMMON-SUBZONE)
output:    (INT2)
subtask  of:   ((DECOMPOSITION     INTERZONAL-DECOMPOSE-METHOD))
conditions:     (NOT  (EQUAL  DEST-SUBZONE     LEAST-COMMON-SUBZONE))

name:    DEFAULT-DEST-COMMON-INT
prototype:     DEFAULT-COMMON-INT
input:    (FINAL-POINT     DEST-SUBZONE     LEAST-COMMON-SUBZONE)
output:    (INT2)
subtask  of:   ((DECOMPOSITION     INTERZONAL-DECOMPOSE-METHOD))
```

conditions: (EQUAL DEST-SUBZONE LEAST-COMMON-SUBZONE)

name: MIDDLE-SUBPROBLEM
prototype: INTRAZONAL-SEARCH
input: (INT1 INT2 LEAST-COMMON-SUBZONE LEAST-COMMON-SUBZONE LEAST-COMMON-SUBZONE)
output: (MIDDLE-PATH)
subtask of: ((DECOMPOSITION INTERZONAL-DECOMPOSE-METHOD))
semantics: (SAME-POINT INITIAL-NODE(MIDDLE-PATH) INT1)
(SAME-POINT FINAL-NODE(MIDDLE-PATH) INT2)

name: SOURCE-SUBPROBLEM
prototype: SEARCH
input: (INITIAL-POINT INT1 INITIAL-ZONE SOURCE-SUBZONE SOURCE-SUBZONE)
output: (FRONT-PATH)
subtask of: ((DECOMPOSITION INTERZONAL-DECOMPOSE-METHOD))

name: DESTINATION-SUBPROBLEM
prototype: SEARCH
input: (INT2 FINAL-POINT DEST-SUBZONE FINAL-ZONE DEST-SUBZONE)
output: (END-PATH)
subtask of: ((DECOMPOSITION INTERZONAL-DECOMPOSE-METHOD))

name: GET-CURRENT-PATH
procedure: GET-CURRENT-PATH
input: (CURRENT-PATHS)
output: (CURRENT-PATH)
subtask of: ((STEP-IN-PATH-INCREASE PATH-INCREASE-METHOD))
semantics: (THERE-IS p IN CURRENT-PATHS: EQUIVALENT-PATH CURRENT-PATH p)

name: EXPAND-CURRENT-PATH
procedure: EXPAND-CURRENT-PATH
input: (CURRENT-PATH A-ZONE)
output: (EXPANDED-PATHS)
subtask of: ((STEP-IN-PATH-INCREASE PATH-INCREASE-METHOD))
semantics: (FOR-ALL p IN EXPANDED-PATHS:
ON-THE-SAME-STREET FINAL-NODE(p) FINAL-NODE(CURRENT-PATH))
(FOR-ALL p IN EXPANDED-PATHS:
EQUIVALENT-PATH PREFIX(p) CURRENT-PATH)

name: SELECT-DESIRED-PATH
procedure: PICK-PATH
input: (EXPANDED-PATHS DESTINATION)
output: (THE-PATH)
subtask of: ((STEP-IN-PATH-INCREASE PATH-INCREASE-METHOD))
semantics: (THERE-IS p IN EXPANDED-PATHS: EQUIVALENT-PATH THE-PATH p)

name: COMBINE-PATH-LISTS
procedure: COMBINE-PATH-LISTS
input: (CURRENT-PATH EXPANDED-PATHS CURRENT-PATHS)
output: (NEW-CURRENT-PATHS)
subtask of: ((STEP-IN-PATH-INCREASE PATH-INCREASE-METHOD))

METHODS

name: ROUTE-PLANNING-METHOD

```
applied to:   ROUTE-PLANNING
subtasks:     (CLASSIFICATION   PATH-RETRIEVAL   SEARCH   STORE)
control:      (''SERIAL-OP''   CLASSIFICATION   PATH-RETRIEVAL   SEARCH   STORE)

name:   TRIVIAL-ROUTE-PLANNING-METHOD
applied to:   ROUTE-PLANNING
subtasks:     (TRIVIAL-ROUTE-PLANNING)
control:      (''SERIAL-OP''   TRIVIAL-ROUTE-PLANNING)
conditions:   (SAME-POINT   INITIAL-POINT   FINAL-POINT)

name:   PATH-RETRIEVAL-METHOD
applied to:   PATH-RETRIEVAL
subtasks:     (RETRIEVAL   FIND-DECOMPOSITION-INDS)
control:      (''SERIAL-OP''   RETRIEVAL   FIND-DECOMPOSITION-INDS)

name:   INTRAZONAL-SEARCH-METHOD
applied to:   SEARCH
subtasks:     (INTRAZONAL-SEARCH)
control:      (''SERIAL-OP''   INTRAZONAL-SEARCH)
conditions:   (ZONE-INTERSECTIONS   INITIAL-ZONE   FINAL-POINT)

name:   MODEL-BASED-METHOD
applied to:   SEARCH
subtasks:     (INTRAZONAL-SEARCH   INTERZONAL-SEARCH)
control:      (''SERIAL-OP''   INTRAZONAL-SEARCH   INTERZONAL-SEARCH)

name:   CASE-BASED-METHOD
applied to:   SEARCH
subtasks:     (PATCH-FRONT   PATCH-END   CBR-PATH-SYNTHESIS)
control:      (''SERIAL-OP''   PATCH-FRONT   PATCH-END   CBR-PATH-SYNTHESIS)
conditions:   ((NOT  ZONE-INTERSECTIONS)   INITIAL-ZONE   FINAL-POINT)
       (NOT  (NULL  MIDDLE-PATH))

name:   INTERZONAL-SEARCH-METHOD
applied to:   SEARCH
subtasks:     (DECOMPOSITION   MBR-PATH-SYNTHESIS)
control:      (''SERIAL-OP''   DECOMPOSITION   MBR-PATH-SYNTHESIS)
conditions:   ((NOT  ZONE-INTERSECTIONS)   INITIAL-ZONE   FINAL-POINT)

name:   INTRAZONAL-SEARCH-METHOD
applied to:   INTRAZONAL-SEARCH
subtasks:     (INIT-SEARCH   PATH-INCREASE)
control:      (''SERIAL-OP''   INIT-SEARCH
       (''LOOP-OP   EXIT-CONDITIONS:   NOT(NULL(DESIRED-PATH))''
       PATH-INCREASE))
conditions:   ((NOT  (EQUAL  INITIAL-POINT   FINAL-POINT)))

name:   INTRAZONAL-METHOD-TRIVIAL
applied to:   INTRAZONAL-SEARCH
subtasks:     (INTRAZONAL-TRIVIAL)
control:      (''SERIAL-OP''   INTRAZONAL-TRIVIAL)
conditions:   (SAME-POINT   INITIAL-POINT   FINAL-POINT)

name:   INTERZONAL-DECOMPOSE-METHOD
applied to:   DECOMPOSITION
```

```
subtasks:    (FIND-LEAST-COMMON-SUBZONE
       FIND-SOURCE-SUBZONE      FIND-SOURCE-COMMON-INTS
       DEFAULT-SOURCE-COMMON-INT
       FIND-DEST-SUBZONE      FIND-DEST-COMMON-INTS
       DEFAULT-DEST-COMMON-INT
       MIDDLE-SUBPROBLEM      SOURCE-SUBPROBLEM      DESTINATION-SUBPROBLEM)
control:    (''SERIAL-OP''     FIND-LEAST-COMMON-SUBZONE
       (''PARALLEL-OP    ORDER:  (1  2)''
          (''SERIAL-OP''     FIND-SOURCE-SUBZONE      FIND-SOURCE-COMMON-INTS
             DEFAULT-SOURCE-COMMON-INT)
          (''SERIAL-OP''     FIND-DEST-SUBZONE      FIND-DEST-COMMON-INTS
             DEFAULT-DEST-COMMON-INT))
       MIDDLE-SUBPROBLEM
       SOURCE-SUBPROBLEM
       DESTINATION-SUBPROBLEM)

name:    PATH-INCREASE-METHOD
applied  to:   STEP-IN-PATH-INCREASE
subtasks:    (GET-CURRENT-PATH      EXPAND-CURRENT-PATH
       SELECT-DESIRED-PATH      COMBINE-PATH-LISTS)
control:    (''SERIAL-OP''     GET-CURRENT-PATH      EXPAND-CURRENT-PATH

     SELECT-DESIRED-PATH      COMBINE-PATH-LISTS)
```

DOMAIN CONCEPTS

```
name:    D-INTERSECTION
domain:    INTERSECTION-DOMAIN
identity  test:    SAME-POINT
attribute:    (STREETS   (LIST-OF   2 D-STREET)   (LAMBDA   (X)  X))

name:    D-ZONE
domain:    ZONE-DOMAIN
identity  test:    EQUALP

name:    D-STREET
domain:    STREET-DOMAIN
identity  test:    EQUALP

name:    D-DIRECTION
domain:    DIRECTION-DOMAIN
identity  test:    EQUAL

name:    D-PATH
domain:    PATH-DOMAIN
identity  test:    EQUIVALENT-PATH
attribute:    (LENGTH  D-NUMBER   (LAMBDA   (X)  (LENGTH   (SIMULATOR-PATH-EQUIVALENT       X))))
attribute:    (REAL-LENGTH    D-NUMBER   SLENGTH)
attribute:    (PREFIX   D-PATH   PATH-PREFIX)
attribute:    (NODES   (LIST-OF   D-INTERSECTION)    (LAMBDA   (X)  X))
attribute:    (INITIAL-NODE    D-INTERSECTION    (LAMBDA   (X)  (CAR  X)))
attribute:    (FINAL-NODE    D-INTERSECTION    (LAMBDA   (X)  (CAR  (LAST  X))))
attribute:    (EDGES   (LIST-OF   D-SEGMENT)    (LAMBDA   (X)  (SEGMENTS   X)))
attribute:    (INITIAL-EDGE    D-SEGMENT   (LAMBDA   (X)  (FIRST   (SEGMENTS   X))))
attribute:    (FINAL-EDGE    D-SEGMENT   (LAMBDA   (X)  (FIRST   (LAST   (SEGMENTS   X)))))
```

```
name:    D-SEGMENT
domain:    PATH-DOMAIN
identity  test:    (LAMBDA  (I J)
   (AND (SAME-POINT    (FIRST  I)  (FIRST  J))  (SAME-POINT    (SECOND  I)  (SECOND  J))))
attribute:      (LENGTH  D-NUMBER  (LAMBDA  (X)  (LENGTH  (SIMULATOR-PATH-EQUIVALENT        X))))
attribute:      (REAL-LENGTH    D-NUMBER  SLENGTH)
attribute:      (PREFIX  D-PATH  PATH-PREFIX)
attribute:      (NODES  (LIST-OF  D-INTERSECTION)    (LAMBDA  (X)  X))
attribute:      (INITIAL-NODE    D-INTERSECTION    (LAMBDA  (X)  (CAR  X)))
attribute:      (FINAL-NODE    D-INTERSECTION    (LAMBDA  (X)  (CAR  (LAST  X))))
attribute:      (EDGES  (LIST-OF  D-SEGMENT)    (LAMBDA  (X)  (SEGMENTS  X)))
attribute:      (INITIAL-EDGE    D-SEGMENT    (LAMBDA  (X)  (FIRST  (SEGMENTS  X))))
attribute:      (FINAL-EDGE    D-SEGMENT    (LAMBDA  (X)  (FIRST  (LAST  (SEGMENTS  X)))))
```

INFORMATION TYPES

```
name:    DESIRED-PATH
type of object:    D-PATH
produced  by:    (ROUTE-PLANNING    SEARCH  TRIVIAL-ROUTE-PLANNING    SEARCH
       INTRAZONAL-SEARCH    CBR-PATH-SYNTHESIS    RECOMPOSITION    MBR-PATH-SYNTHESIS
       PATH-INCREASE    INTRAZONAL-TRIVIAL)
input  to:    (STORE)

name:    THE-PATH
type of object:    D-PATH
produced  by:    (STEP-IN-PATH-INCREASE        SELECT-DESIRED-PATH)

name:    POSSIBLE-PATHS
type of object:    D-PATH
produced  by:    (INIT-SEARCH    PATH-INCREASE)
input  to:    (PATH-INCREASE)

name:    CURRENT-PATHS
type of object:    D-PATH
input  to:    (STEP-IN-PATH-INCREASE        GET-CURRENT-PATH    COMBINE-PATH-LISTS)

name:    NEW-CURRENT-PATHS
type of object:    D-PATH
produced  by:    (STEP-IN-PATH-INCREASE        COMBINE-PATH-LISTS)

name:    CURRENT-PATH
type of object:    D-PATH
produced  by:    (GET-CURRENT-PATH)
input  to:    (EXPAND-CURRENT-PATH        COMBINE-PATH-LISTS)

name:    PATH-TO-GOAL
type of object:    D-PATH

name:    EXPANDED-PATHS
type of object:    D-PATH
produced  by:    (EXPAND-CURRENT-PATH)
input  to:    (SELECT-DESIRED-PATH        COMBINE-PATH-LISTS)

name:    FRONT-PATH
```

type of object: D-PATH
produced by: (PATCH-FRONT DECOMPOSITION SOURCE-SUBPROBLEM)
input to: (CBR-PATH-SYNTHESIS RECOMPOSITION MBR-PATH-SYNTHESIS)

name: END-PATH
type of object: D-PATH
produced by: (PATCH-END DECOMPOSITION DESTINATION-SUBPROBLEM)
input to: (CBR-PATH-SYNTHESIS RECOMPOSITION MBR-PATH-SYNTHESIS)

name: MIDDLE-PATH
type of object: D-PATH
produced by: (PATH-RETRIEVAL RETRIEVAL RETRIEVAL-PROTOTYPE DECOMPOSITION
MIDDLE-SUBPROBLEM)
input to: (FIND-DECOMPOSITION-INITS SEARCH RECOMPOSITION DECOMPOSITION
CBR-PATH-SYNTHESIS MBR-PATH-SYNTHESIS)

name: A-ZONE
type of object: D-ZONE
input to: (STEP-IN-PATH-INCREASE EXPAND-CURRENT-PATH)

name: SUBZONE
type of object: D-ZONE
input to: (FIND-COMMON-INITS DEFAULT-COMMON-INIT)

name: COMMON-SUBZONE
type of object: D-ZONE
input to: (FIND-COMMON-INITS DEFAULT-COMMON-INIT)

name: TOP-NEIGHBORHOOD
type of object: D-ZONE
input to: (ROUTE-PLANNING CLASSIFICATION SEARCH PATCH-FRONT PATCH-END DECOMPOSITION)

name: INITIAL-ZONE
type of object: D-ZONE
produced by: (CLASSIFICATION)
input to: (PATH-RETRIEVAL STORE RETRIEVAL RETRIEVAL-PROTOTYPE SEARCH
INTRAZONAL-SEARCH DECOMPOSITION PATH-INCREASE FIND-LEAST-COMMON-SUBZONE
FIND-SOURCE-SUBZONE SOURCE-SUBPROBLEM)

name: FINAL-ZONE
type of object: D-ZONE
produced by: (CLASSIFICATION)
input to: (PATH-RETRIEVAL STORE RETRIEVAL RETRIEVAL-PROTOTYPE SEARCH
DECOMPOSITION FIND-LEAST-COMMON-SUBZONE FIND-DEST-SUBZONE DESTINATION-SUBPROBLEM)

name: SOURCE-SUBZONE
type of object: D-ZONE
produced by: (FIND-SOURCE-SUBZONE)
input to: (FIND-SOURCE-COMMON-INITS DEFAULT-SOURCE-COMMON-INIT SOURCE-SUBPROBLEM)

name: DEST-SUBZONE
type of object: D-ZONE
produced by: (FIND-DEST-SUBZONE)
input to: (FIND-DEST-COMMON-INITS DEFAULT-DEST-COMMON-INIT DESTINATION-SUBPROBLEM)

name: LEAST-COMMON-SUBZONE
type of object: D-ZONE
produced by: (FIND-LEAST-COMMON-SUBZONE)
input to: (FIND-SOURCE-SUBZONE FIND-SOURCE-COMMON-INIS DEFAULT-SOURCE-COMMON-INT
FIND-DEST-SUBZONE FIND-DEST-COMMON-INIS DEFAULT-DEST-COMMON-INT MIDDLE-SUBPROBLEM)

name: INITIAL-POINT
type of object: D-INTERSECTION
input to: (ROUTE-PLANNING CLASSIFICATION PATH-RETRIEVAL STORE TRIVIAL-ROUTE-PLANNING
RETRIEVAL RETRIEVAL-PROTOTYPE SEARCH INTRAZONAL-SEARCH PATCH-FRONT
DECOMPOSITION INIT-SEARCH INTRAZONAL-TRIVIAL FIND-SOURCE-COMMON-INIS
DEFAULT-SOURCE-COMMON-INT SOURCE-SUBPROBLEM)

name: FINAL-POINT
type of object: D-INTERSECTION
input to: (ROUTE-PLANNING CLASSIFICATION PATH-RETRIEVAL STORE TRIVIAL-ROUTE-PLANNING
RETRIEVAL RETRIEVAL-PROTOTYPE SEARCH INTRAZONAL-SEARCH PATCH-END
DECOMPOSITION PATH-INCREASE INTRAZONAL-TRIVIAL FIND-DEST-COMMON-INIS
DEFAULT-DEST-COMMON-INT DESTINATION-SUBPROBLEM)

name: DESTINATION
type of object: D-INTERSECTION
input to: (STEP-IN-PATH-INCREASE SELECT-DESIRED-PATH)

name: INT1
type of object: D-INTERSECTION
produced by: (PATH-RETRIEVAL FIND-DECOMPOSITION-INIS FIND-SOURCE-COMMON-INIS
DEFAULT-SOURCE-COMMON-INT)
input to: (PATCH-FRONT CER-PATH-SYNTHESIS RECOMPOSITION DECOMPOSITION
MER-PATH-SYNTHESIS MIDDLE-SUBPROBLEM SOURCE-SUBPROBLEM)

name: INT2
type of object: D-INTERSECTION
produced by: (PATH-RETRIEVAL FIND-DECOMPOSITION-INIS FIND-DEST-COMMON-INIS
DEFAULT-DEST-COMMON-INT)
input to: (PATCH-END CER-PATH-SYNTHESIS RECOMPOSITION DECOMPOSITION
MER-PATH-SYNTHESIS MIDDLE-SUBPROBLEM DESTINATION-SUBPROBLEM)

name: ANCHOR-INT
type of object: D-INTERSECTION
input to: (FIND-COMMON-INIS DEFAULT-COMMON-INT)

name: COMMON-INT
type of object: D-INTERSECTION
produced by: (FIND-COMMON-INIS DEFAULT-COMMON-INT)

DOMAIN RELATIONS

name: LESS-THAN
input args: (D-NUMBER D-NUMBER)
predicate: <

name: GREATER-THAN
input args: (D-NUMBER D-NUMBER)

```
predicate:      >

name:    LESS-EQUAL
input  args:    (D-NUMBER    D-NUMBER)
predicate:      < =

name:    GREATER-EQUAL
input  args:    (D-NUMBER    D-NUMBER)
predicate:      > =

name:    CHILDREN-ZONES
input  args:    (D-ZONE)
output  args:    ((LIST-OF    D-ZONE))
truth  table:    CHILDREN-ZONES-RELATION
indexing   relation:     T

name:    ZONE-INTERSECTIONS
input  args:    (D-ZONE)
output  args:    ((LIST-OF    D-INTERSECTION))
truth  table:    ZONE-INTERSECTIONS-RELATION
indexing   relation:     T

name:    ZONES-OF-INT
input  args:    (D-INTERSECTION)
output  args:    ((LIST-OF    D-ZONE))
truth  table:    ZONES-OF-INT-RELATION
indexing   relation:     T

name:    LANDMARK-LOCATION
input  args:    (D-INTERSECTION)
predicate:      (LAMBDA    (X)  (IS-IT-A-LANDMARK      X))
indexing   relation:     T

name:    ZONE-STREETS
input  args:    (D-ZONE)
output  args:    ((LIST-OF    D-STREET))
truth  table:    ZONE-STREETS-RELATION
indexing   relation:     T

name:    ON-THE-SAME-STREET
input  args:    (D-INTERSECTION)
output  args:    ((LIST-OF    D-INTERSECTION))
predicate:      (LAMBDA    (X)  (ADJACENT-INTS      X))
inverse  predicate:      (LAMBDA    (X)  (ADJACENT-INTS      X))

name:    LEGAL-DIRECTION
input  args:    (D-STREET)
output  args:    ((LIST-OF    D-DIRECTION))
truth  table:    LEGAL-DIRECTION-RELATION

name:    FIRST-SUBPATH
input  args:    (D-PATH)
output  args:    ((LIST-OF    D-PATH))
predicate:      (LAMBDA    (X)  (FIRST-SUBPATH      X))
```

```
name:     SECOND-SUBPATH
input args:     (D-PATH)
output args:     ((LIST-OF D-PATH))
predicate:     (LAMBDA (X) (SECOND-SUBPATH X))

name:     THIRD-SUBPATH
input args:     (D-PATH)
output args:     ((LIST-OF D-PATH))
predicate:     (LAMBDA (X) (THIRD-SUBPATH X))
```

DOMAIN CONSTRAINTS

```
relation1:     ZONE-INTERSECTIONS
relation2:     ZONES-OF-INT
relation1:     ZONES-OF-INT
relation2:     ZONE-INTERSECTIONS
```

APPENDIX B

THE SBF MODEL OF KRITIK'S ADAPTIVE DESIGN

TASKS

```
name:    DESIGN
methods:     (DESIGN-METHOD)
input:     (NEW-FUNCTION)
output:     (NEW-CASE)
semantics:     ((NOT  NULL)  NEW-CASE  MODEL)
       ((NOT  NULL)  NEW-CASE  STRUCTURE)
       (SAME-FUNCTION     NEW-CASE  FUNCTION  NEW-FUNCTION     NIL)

name:    RETRIEVAL
methods:     (SUCCESSIVE-REFINEMENT-SEARCH-METHOD)
input:     (NEW-FUNCTION)
output:     (OLD-CASE)
subtask  of:   ((DESIGN  DESIGN-METHOD))

name:    ADAPTATION
methods:     (MODEL-BASED-ADAPTATION-METHOD)
input:     (OLD-CASE  NEW-FUNCTION)
output:     (NEW-CASE)
subtask  of:   ((DESIGN  DESIGN-METHOD))
conditions:     ((NOT  SAME-FUNCTION)     NEW-FUNCTION     FUNCTION(OLD-CASE))

name:    CASE-SELECTION
methods:     (PROPERTY-VALUE-REFINEMENT-METHOD)
input:     (NEW-FUNCTION)
output:     (LIST-OF-CASES)
subtask  of:   ((RETRIEVAL     SUCCESSIVE-REFINEMENT-SEARCH-METHOD))

name:    CASE-ORDERING
procedure:     NEW-ORDER-CASES
input:     (NEW-FUNCTION     LIST-OF-CASES)
output:     (OLD-CASE)
subtask  of:   ((RETRIEVAL     SUCCESSIVE-REFINEMENT-SEARCH-METHOD))
semantics:     (THERE-IS   i IN LIST-OF-CASES:     EQUALP(OLD-CASE     i))

name:    FUNCTIONAL-DIFFERENCES-IDENTIFICATION
procedure:     FIND-FUNC-SPEC-DIFF
input:     (OLD-CASE  NEW-FUNCTION)
output:     (LIST-OF-DIFFS)
subtask  of:   ((ADAPTATION     MODEL-BASED-ADAPTATION-METHOD))

name:    BLAME-ASSIGNMENT
```

```
procedure:    DIAGNOSE
input:    (OLD-CASE    LIST-OF-DIFFS)
output:    (LIST-OF-SUSPICIOUS-COMPONENTS)
subtask    of:    ((ADAPTATION    MODEL-BASED-ADAPTATION-METHOD))

name:    REPAIR
methods:    (COMPONENT-REPLACEMENT-PLAN    STRUCTURE-REPLICATION-PLAN)
input:    (OLD-CASE    NEW-FUNCTION    LIST-OF-DIFFS    LIST-OF-SUSPICIOUS-COMPONENTS)
output:    (NEW-BEHAVIOR    NEW-STRUCTURE)
subtask    of:    ((ADAPTATION    MODEL-BASED-ADAPTATION-METHOD))

name:    MODEL-ASSEMBLY
procedure:    ASSEMBLE
input:    (NEW-FUNCTION    NEW-BEHAVIOR    NEW-STRUCTURE)
output:    (NEW-CASE)
subtask    of:    ((ADAPTATION    MODEL-BASED-ADAPTATION-METHOD))
semantics:    (SAME-FUNCTION    FUNCTION(NEW-CASE)    NEW-FUNCTION)

name:    PROPERTY-REFINEMENT
methods:    (SELECTION-OF-RELEVANT-PROPERTY-NODES)
input:    (NEW-FUNCTION)
output:    (PROPERTY-NODES)
subtask    of:    ((CASE-SELECTION    PROPERTY-VALUE-REFINEMENT-METHOD))

name:    VALUE-REFINEMENT
methods:    (SELECTION-OF-RELEVANT-VALUE-NODES)
input:    (NEW-FUNCTION    PROPERTY-NODES)
output:    (MEMORY-NODES    PROPERTY-NODES)
subtask    of:    ((CASE-SELECTION    PROPERTY-VALUE-REFINEMENT-METHOD))

name:    GET-CASES-OF-NODE
prototype:    T-GET-CASES-PROTOTYPE
input:    (A-NODE)
output:    (ITS-CASES)
subtask    of:    ((CASE-SELECTION    PROPERTY-VALUE-REFINEMENT-METHOD))

name:    COMPONENT-REIRIEVAL
methods:    (COMPONENT-REIRIEVAL-METHOD)
input:    (LIST-OF-SUSPICIOUS-COMPONENTS    LIST-OF-DIFFS)
output:    (COMPONENT-TO-REPLACE    REPLACEMENT)
subtask    of:    ((REPAIR    COMPONENT-REPLACEMENT-PLAN))

name:    COMPONENT-REPLACEMENT
procedure:    APPLY-COMPONENT-REPLACEMENT
input:    (OLD-CASE    COMPONENT-TO-REPLACE    REPLACEMENT    NEW-FUNCTION    LIST-OF-DIFFS)
output:    (NEW-BEHAVIOR    NEW-STRUCTURE)
subtask    of:    ((REPAIR    COMPONENT-REPLACEMENT-PLAN))

name:    IDENTIFY-COMPONENT-TO-REPLICATE
procedure:    FIND-COMPONENT-TO-REPLICATE
input:    (LIST-OF-SUSPICIOUS-COMPONENTS    LIST-OF-DIFFS)
output:    (COMPONENT-TO-REPLICATE)
subtask    of:    ((REPAIR    STRUCTURE-REPLICATION-PLAN))

name:    STRUCTURE-REPLICATION
```

```
procedure:      APPLY-CASCADE
input:      (OLD-CASE   COMPONENT-TO-REPLICATE)
output:      (NEW-BEHAVIOR      NEW-STRUCTURE)
subtask   of:    ((REPAIR   STRUCTURE-REPLICATION-PLAN))

name:     IDENTIFY-NEW-FUNCTION's-PROPERTIES
procedure:      NEW-GET-PROPS
input:      (NEW-FUNCTION)
output:      (NEW-FUNCTION-PROPS)
subtask   of:    ((PROPERTY-REFINEMENT        SELECTION-OF-RELEVANT-PROPERTY-NODES)          )

name:     IDENTIFY-KNOWN-PROPERTIES
procedure:      KEEP-PROPS-THAT-ARE-CURRENT-INDICES
input:      (NEW-FUNCTION-PROPS)
output:      (MATCHING-PROPS)
subtask   of:    ((PROPERTY-REFINEMENT        SELECTION-OF-RELEVANT-PROPERTY-NODES)          )

name:     PROBLEM-ELABORATION
procedure:      ELABORATE-PROBLEM-DESCRIPTION
input:      (NEW-FUNCTION      MATCHING-PROPS)
output:      (MATCHING-PROPS)
subtask   of:    ((PROPERTY-REFINEMENT        SELECTION-OF-RELEVANT-PROPERTY-NODES)          )

name:     REFINEMENT-ALONG-PROPERTIES
procedure:      GET-ALL-MATCHING-PROPS
input:      (MATCHING-PROPS)
output:      (PROPERTY-NODES)
subtask   of:    ((PROPERTY-REFINEMENT        SELECTION-OF-RELEVANT-PROPERTY-NODES)          )

name:     IDENTIFY-VALUES
procedure:      GET-RELEVANT-VALUES
input:      (PROPERTY-NODES      NEW-FUNCTION)
output:      (VALUES)
subtask   of:    ((VALUE-REFINEMENT       SELECTION-OF-RELEVANT-VALUE-NODES))

name:     REFINEMENT-ALONG-VALUES
procedure:      GET-ALL-MATCHING-VALUES
input:      (PROPERTY-NODES      VALUES)
output:      (MEMORY-NODES)
subtask   of:    ((VALUE-REFINEMENT       SELECTION-OF-RELEVANT-VALUE-NODES))

name:     COMPONENT-RETRIEVAL-LOWER-PARAM
procedure:      FIND-REPLACEABLE-COMPONENT-FAULT
input:      (LIST-OF-SUSPICIOUS-COMPONENTS          LIST-OF-DIFFS)
output:      (COMPONENT-TO-REPLACE        REPLACEMENT)
subtask   of:    ((COMPONENT-RETRIEVAL        COMPONENT-RETRIEVAL-METHOD))
semantics:      (THERE-IS    i IN LIST-OF-SUSPICIOUS-COMPONENTS:          EQUALP   COMPONENT-TO-REPLACE      i)
        (EQUALP   PROTOTYPE(REPLACEMENT)        COMPONENT-PROTOTYPE(COMPONENT-TO-REPLACE)          )
        (IS-LESSER-THAN      PARAMETER-VALUE(REPLACEMENT)        PARAMETER-VALUE(COMPONENT-TO-REPLACE))
conditions:      (APPLICABLE-TO-SUBSTANCE-RANGE-DIFF          LIST-OF-DIFFS)
        (DESIRED-RANGE-SMALLER-THAN-CURRENT          LIST-OF-DIFFS)

name:     COMPONENT-RETRIEVAL-HIGHER-PARAM
procedure:      FIND-REPLACEABLE-COMPONENT-FAULT
input:      (LIST-OF-SUSPICIOUS-COMPONENTS          LIST-OF-DIFFS)
```

output: (COMPONENT-TO-REPLACE REPLACEMENT)
subtask of: ((COMPONENT-REIRIEVAL COMPONENT-REIRIEVAL-METHOD))
semantics: (THERE-IS i IN LIST-OF-SUSPICIOUS-COMPONENTS: EQUALP COMPONENT-TO-REPLACE i)
(EQUALP PROTOTYPE(REPLACEMENT) COMPONENT-PROTOTYPE(COMPONENT-TO-REPLACE))
(IS-BIGGER-THAN PARAMETER-VALUE(REPLACEMENT) PARAMETER-VALUE(COMPONENT-TO-REPLACE))
conditions: (APPLICABLE-TO-SUBSTANCE-RANGE-DIFF LIST-OF-DIFFS)
(DESIRED-RANGE-BIGGER-THAN-CURRENT LIST-OF-DIFFS)

METHODS

name: DESIGN-METHOD
applied to: DESIGN
subtasks: (REIRIEVAL ADAPTATION)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP)) REIRIEVAL ADAPTATION)

name: SUCCESSIVE-REFINEMENT-SEARCH-METHOD
applied to: REIRIEVAL
subtasks: (CASE-SELECTION CASE-ORDERING)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP)) CASE-SELECTION CASE-ORDERING)

name: MODEL-BASED-ADAPTATION-METHOD
applied to: ADAPTATION
subtasks: (FUNCTIONAL-DIFFERENCES-IDENTIFICATIO N BLAME-ASSIGNMENT REPAIR MODEL-ASSEMBLY)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP))
FUNCTIONAL-DIFFERENCES-IDENTIFICATIO N BLAME-ASSIGNMENT REPAIR MODEL-ASSEMBLY)

name: PROPERTY-VALUE-REFINEMENT-METHOD
applied to: CASE-SELECTION
subtasks: (PROPERTY-REFINEMENT VALUE-REFINEMENT GET-CASES-OF-NODE)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP))
PROPERTY-REFINEMENT VALUE-REFINEMENT
((MAKE-INSTANCE (QUOTE LOOP-OP) GET-CASES-OF-NODE))

name: COMPONENT-REPLACEMENT-PLAN
applied to: REPAIR
subtasks: (COMPONENT-REIRIEVAL COMPONENT-REPLACEMENT)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP))
COMPONENT-REIRIEVAL COMPONENT-REPLACEMENT)
conditions: (APPLICABLE-TO-SUBSTANCE-RANGE-DIFF LIST-OF-DIFFS)

name: STRUCTURE-REPLICATION-PLAN
applied to: REPAIR
subtasks: (IDENTIFY-COMPONENT-TO-REPLICATE STRUCTURE-REPLICATION)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP))
IDENTIFY-COMPONENT-TO-REPLICATE STRUCTURE-REPLICATION)
conditions: (APPLICABLE-TO-SUBSTANCE-RANGE-DIFF LIST-OF-DIFFS)

name: SELECTION-OF-RELEVANT-PROPERTY-NODES
applied to: PROPERTY-REFINEMENT
subtasks: (IDENTIFY-NEW-FUNCTION's-PROPERTIES IDENTIFY-KNOWN-PROPERTIES
PROBLEM-ELABORATION REFINEMENT-ALONG-PROPERTIES)
control: ((MAKE-INSTANCE (QUOTE SERIAL-OP))
IDENTIFY-NEW-FUNCTION's-PROPERTIES IDENTIFY-KNOWN-PROPERTIES
PROBLEM-ELABORATION REFINEMENT-ALONG-PROPERTIES)

name: SELECTION-OF-RELEVANT-VALUE-NODES
applied to: VALUE-REFINEMENT
subtasks: (IDENTIFY-VALUES REFINEMENT-ALONG-VALUES)
control: ((MAKE-INSTANCE (QUOTE SERIAL-CP))
IDENTIFY-VALUES REFINEMENT-ALONG-VALUES)

name: COMPONENT-RETRIEVAL-METHOD
applied to: COMPONENT-RETRIEVAL
subtasks: (COMPONENT-RETRIEVAL-LOWER-PARAM
COMPONENT-RETRIEVAL-HIGHER-PARAM)
control: ((MAKE-INSTANCE (QUOTE SERIAL-CP))
COMPONENT-RETRIEVAL-LOWER-PARAM
COMPONENT-RETRIEVAL-HIGHER-PARAM)

INFORMATION TYPES

name: NEW-FUNCTION
type of object: D-FUNCTION
input to: (IDENTIFY-VALUES PROBLEM-ELABORATION
IDENTIFY-NEW-FUNCTION's-PROPERTIES COMPONENT-REPLACEMENT
VALUE-REFINEMENT PROPERTY-REFINEMENT MODEL-ASSEMBLY
REPAIR FUNCTIONAL-DIFFERENCES-IDENTIFICATION CASE-ORDERING
CASE-SELECTION ADAPTATION RETRIEVAL DESIGN)

name: LIST-OF-CASES
type of object: D-DESIGN-CASE
produced by: (CASE-SELECTION)
input to: (CASE-ORDERING)

name: ITS-CASES
type of object: D-DESIGN-CASE
produced by: (GET-CASES-OF-NODE)

name: PROPERTY-NODES
type of object: D-MEMORY-NODE
produced by: (REFINEMENT-ALONG-PROPERTIES VALUE-REFINEMENT PROPERTY-REFINEMENT)
input to: (REFINEMENT-ALONG-VALUES IDENTIFY-VALUES VALUE-REFINEMENT)

name: A-NODE
type of object: D-MEMORY-NODE
input to: (GET-CASES-OF-NODE)

name: MEMORY-NODES
type of object: D-MEMORY-NODE
produced by: (REFINEMENT-ALONG-VALUES VALUE-REFINEMENT)

name: NEW-FUNCTION-PROPS
type of object: D-PROPERTY
produced by: (IDENTIFY-NEW-FUNCTION's-PROPERTIES)
input to: (IDENTIFY-KNOWN-PROPERTIES)

name: MATCHING-PROPS
type of object: D-PROPERTY
produced by: (PROBLEM-ELABORATION IDENTIFY-KNOWN-PROPERTIES)
input to: (REFINEMENT-ALONG-PROPERTIES PROBLEM-ELABORATION)

name: ELAB-PROPS
type of object: D-PROPERTY

name: VALUES
type of object: D-VALUE
produced by: (IDENTIFY-VALUES)
input to: (REFINEMENT-ALONG-VALUES)

name: VALUE-NODES
type of object: D-MEMORY-NODE

name: OLD-CASE
type of object: D-DESIGN-CASE
produced by: (CASE-ORDERING RETRIEVAL)
input to: (STRUCTURE-REPLICATION COMPONENT-REPLACEMENT REPAIR
BLAME-ASSIGNMENT FUNCTIONAL-DIFFERENCES-IDENTIFICATION ADAPTATION)

name: NEW-CASE
type of object: D-DESIGN-CASE
produced by: (MODEL-ASSEMBLY ADAPTATION DESIGN)

name: NEW-BEHAVIOR
type of object: D-BEHAVIOR
produced by: (STRUCTURE-REPLICATION COMPONENT-REPLACEMENT REPAIR)
input to: (MODEL-ASSEMBLY)

name: NEW-STRUCTURE
type of object: D-STRUCTURE
produced by: (STRUCTURE-REPLICATION COMPONENT-REPLACEMENT REPAIR)
input to: (MODEL-ASSEMBLY)

name: LIST-OF-DIFFS
type of object: D-DIFFERENCE
produced by: (FUNCTIONAL-DIFFERENCES-IDENTIFICATION)
input to: (IDENTIFY-COMPONENT-TO-REPLICATE COMPONENT-REPLACEMENT
COMPONENT-RETRIEVAL-LOWER-PARAM
COMPONENT-RETRIEVAL-HIGHER-PARAM
COMPONENT-RETRIEVAL REPAIR BLAME-ASSIGNMENT)

name: LIST-OF-SUSPICIOUS-COMPONENTS
type of object: D-SUSPICIOUS-COMP
produced by: (BLAME-ASSIGNMENT)
input to: (IDENTIFY-COMPONENT-TO-REPLICATE
COMPONENT-RETRIEVAL-LOWER-PARAM
COMPONENT-RETRIEVAL-HIGHER-PARAM
COMPONENT-RETRIEVAL REPAIR)

name: COMPONENT-TO-REPLACE
type of object: D-SUSPICIOUS-COMP
produced by: (COMPONENT-RETRIEVAL-LOWER-PARAM
COMPONENT-RETRIEVAL-HIGHER-PARAM
COMPONENT-RETRIEVAL)
input to: (COMPONENT-REPLACEMENT)

```
name:    REPLACEMENT
type of object:    D-COMPONENT
produced by:    (COMPONENT-RETRIEVAL-LOWER-PARAM
       COMPONENT-RETRIEVAL-HIGHER-PARAM
       COMPONENT-RETRIEVAL)
input to:    (COMPONENT-REPLACEMENT)

name:    COMPONENT-TO-REPLICATE
type of object:    D-SUSPICIOUS-COMP
produced by:    (IDENTIFY-COMPONENT-TO-REPLICATE)
input to:    (STRUCTURE-REPLICATION)
```

DOMAIN RELATIONS

```
name:    MORE-ABSTRACT
input args:    (D-SUBSTANCE-CONCEPT      D-SUBSTANCE-CONCEPT)
predicate:     (LAMBDA   (X Y)
       (OR (EQUAL   (SLOT-VALUE    X (QUOTE   NAME))   (SLOT-VALUE    Y (QUOTE   NAME)))
         (MEMBER   (SLOT-VALUE    Y (QUOTE   NAME))
           (SUBSTANCE-SPECIALIZATIONS        (SLOT-VALUE    X (QUOTE   NAME))))))

name:    LESS-ABSTRACT
input args:    (D-SUBSTANCE-CONCEPT      D-SUBSTANCE-CONCEPT)
predicate:     (LAMBDA   (X Y)
       (OR (EQUAL   (SLOT-VALUE    X (QUOTE   NAME))   (SLOT-VALUE    Y (QUOTE   NAME)))
         (MEMBER   (SLOT-VALUE    X (QUOTE   NAME))
           (SUBSTANCE-SPECIALIZATIONS        (SLOT-VALUE    Y (QUOTE   NAME))))))

name:    ROOT-SPECIALIZATION
input args:    (D-MEMORY-ROOT     D-PROPERTY)
output args:    (LIST-OF    D-MEMORY-NODE)
truth table:    ROOT-SPECIALIZATION-RELATION
indexing relation:     T

name:    NODE-SPECIALIZATION
input args:    (D-MEMORY-NODE     D-VALUE)
output args:    (LIST-OF    D-MEMORY-NODE)
truth table:    NODE-SPECIALIZATION-RELATION
indexing relation:     T

name:    VALUE-SPECIALIZATION
input args:    (D-VALUE)
output args:    (LIST-OF    D-VALUE)
truth table:    VALUE-SPECIALIZATION-RELATION

name:    INDEXING
input args:    (D-MEMORY-NODE)
output args:    (D-DESIGN-CASE)
truth table:    INDEXING-RELATION
indexing relation:     T
```

DOMAIN CONSTRAINTS

Bibliography

[Abu-Hanna *et al.* 1991] Abu-Hanna, A., Benjamins, V.R., and Jansweijer, W.N.H. (1991) Device Understanding and modeling for Diagnosis. In *IEEE EXPERT*, 6(2):26-32.

[Alexander *et al.* 1989] Alexander, P., Minden, G., Tsatsoulis, C., and Holtzman, J. (1989) Storing Design Knowledge in Cases. In *Proceedings Second DARPA Workshop on Case-Based Reasoning* pp. 188-192.

[Allemang 1990] Allemang, D. (1990) Understanding Programs as Devices, Ph.D. Dissertation, The Ohio State University.

[Amarel 1968] Amarel, S. (1968) On representation of problems of reasoning about actions. *Machine Intelligence 3*, D. Michie (ed.), pp. 131-171, Edinburgh University Press.

[Anderson, 1982] Anderson, J. (1982) Acquisition of cognitive skill. *Psychology Review*, 89:369-406.

[Arkin 1986] Arkin, R.C. (1986) Path Planning for a Vision-Based Autonomous Robot, Univ. of Mass. at Amherst, Report COINS 86-48.

[Baker and Brown 1981] Baker, L., and Brown, A.L. (1981) Metacognition and the reading process. In A Handbook of reading research, ed. D. Pearson, pp. 353-394. New York: Plenum Press.

[Barletta and Mark 1988] Barletta, R., and Mark, W. (1988) Explanation-Based Indexing of Cases. *Proceedings Seventh National Conference on Artificial Intelligence*, 541-546.

[Barr 1979] Barr, A. (1979) Meta-Knowledge and Cognition. In *Proceedings of the Sixth International Joint Conference on AI*.

[Bhatta and Goel 1992] Bhatta, S. and Goel, A. (1992) Use of Mental Models for Constraining Index Learning in Experience-Based Design, In Proceedings of the AAAI workshop on Constraining Learning with Prior Knowledge, pp. 1-10, San Jose, CA.

[Bhatta and Goel 1993] Bhatta, S. and Goel, A. (1993) Model-Based Learning of Structural Indices to Design Cases, In *Proceedings of the International Joint Conference on AI* Workshop on "Reuse of Designs: An Interdisciplinary Cognitive Approach", pp. 1-13, Chambery, Savoie, France.

[Bhatta and Goel 1994] Bhatta, S. and Goel, A. (1993) Discovery of Physical Principles from Design Experiences, In *International Journal of AI EDAM (AI in Engineering Design, Analysis, and Manufacturing*, 8(2).

[Bhatta 1995] Bhatta, S. (1995) A Model-Based approach to analogical reasoning and learning to design, PhD Thesis, Georgia Institute of Technology (forthcoming).

[Brown 1987] Brown, A. (1987) Metacognition, Executive Control, Self-Regulation, and Other more Mysterious Mechanisms. In F.E. Weinert and R.H. Kluwe (eds.), *Metacognition, Motivation, and Understanding*, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Brown and Chandrasekaran 1989] Brown, D., and Chandrasekaran, B. (1989) *Design Problem Solving: Knowledge Structures and Control Strategies*, San Mateo, CA: Morgan Kaufmann.

[Bylander and Chandrasekaran 1985] Bylander, T. and Chandrasekaran, B. (1985) Understanding Behavior Using Consolidation, In *Proceedings of the Ninth International Joint Conference on AI*, pp. 450-454.

[Bylander and Chandrasekaran 1987] Bylander, T., and Chandrasekaran, B. (1987) Generic tasks for knowledge-based reasoning: the 'right' level of abstraction for knowledge acquisition. *Int. J. Man-Mach. Stud. (UK)* vol.26, no.2 p.231-43.

[Carbonell 1986] Carbonell, J.G. (1986) Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In R. Michalski, J. Carbonell and T. Mitchell (eds): Machine Learning: An Artificial Intelligence Approach, Volume II, San Mateo, CA: Morgan Kauffman.

[Carbonell *et al.* 1989] Carbonell, J.G., Knoblock, C.A., and Minton, S. (1989) Prodigy: An Integrated Architecture for Planning and Learning. In *Architectures for Intelligence*, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Catrambone 1993a] Catrambone, R. (in press). Aiding subgoal learning: Effects on transfer. *Journal of Educational Psychology*.

[Catrambone 1993b] Catrambone, R. (in press). Improving examples to improve transfer to novel problems. Memory and Cognition.

[Chandrasekaran 1983] Chandrasekaran, B. (1983) Towards a taxonomy of of problem-solving types, *AI magazine* 4(1):9-17.

[Chandrasekaran 1987] Chandrasekaran, B. (1987) Towards a functional architecture for intelligence based on generic information processing tasks. In *Proceedings of the Tenth International Joint Conference on AI*, pp. 1183-1192, Milan, Italy. San Mateo, CA: Morgan Kauffmann Publishers.

[Chandrasekaran *et al.* 1989] Chandrasekaran, B., Tanner, M.C., and Josephson, J.R. (1989) Explaining control strategies in problem solving *IEEE Expert* vol.4, no.1 pp.9-15, 19-24.

[Redmond 1992] Redmond, M. (1993) Learning by Observing and Understanding Expert Problem Solving. Ph.D. Dissertation, Georgia Institute of Technology, GIT-CC-92/43.

[Rosenbloom *et al.* 1989] Rosenbloom, P.S., Newell, A., and Laird, J.E. (1989) Toward the Knowledge Level in Soar: Architecture in the Use of Knowledge. In K. VanLehn (ed.): Architectures for Intelligence, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Chandrasekaran and Goel 1988] Chanrasekaran, B. and Goel, A. (1988) From Numbers to Symbols to Knowledge Structures: Artificial Intelligence Perspectives on the Classification Task. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(3):415-424, IEEE Press.

[Chandrasekaran 1989] Chandrasekaran, B. (1989) Task Structures, Knowledge Acquisition and Machine Learning. Machine Learning 4, 339-345.

[Chandrasekaran and Johnson 1993] Chandrasekaran, B. and Johnson, T.R. (1993) Generic Tasks and Task Structures: History, Critique and new Directions. In D. Krivine and R. Simmons (eds.): Second Generation Expert Systems, Springer-Verlag, pp. 232-272.

[Chandrasekaran *et. al* 1993] Chandrasekaran, B., Goel, A., and Iwasaki, Y. (1993) Functional Representation As Design Rationale, *IEEE Computer*, January 1993, 48-56.

[Chi 1987] Chi, M. (1987) Representing knowledge and metaknowledge: implications for interpreting metamemory research, In F.E. Weinert and R.H. Kluwe (eds.), *Metacognition, Motivation, and Understanding*, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Clancey 1985] Clancey, W.J. (1985) Heuristic Classification, *Artificial Intelligence* 27:289:350.

[Clancey 1986] Clancey, W.J. (1986) From GUIDON to NEOMYCIN and HERACLES in twenty short lessons, Stanford University Report STAN-CS-87-1172; also, KSL-86-11

[AI Hhanbook(III)] The Handbook of Artificial Intelligence, P.R. Cohen and E.A. Feigenbaum (eds.) Adison Wesley 1989.

[Davis 1977] Davis, R. (1977) Interactive transfer of expertise: Acquisition of new inference rules, *Artificial Intelligence* 12:121-157.

[Davis and Buchanan 1977] Davis, R., and Buchanan, B. (1977) Meta-Level Knowledge: Overview and Applications. In *Proceedings of the Fifth International Joint Conference on AI.*

[Davis 1980] Davis, R. (1980) Meta-Rules: Reasoning about Control. *Artificial Intelligence* 15:179-222.

[Davis 1984] Davis, R. (1984) Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24:347-410.

[Davis and Hamscher 1988] Davis, R. and Hamscher, W. (1988) Model Based Troubleshooting. *Exploring Artificial Intelligence*, H. Shrobe (editor). San Mateo, CA: Morgan Kaufmann.

[Daube and Hayes-Roth 1989] Daube, F., and Hayes-Roth, B. (1989) A Case-Based Mechanical Redesign System. *Proceedings Eleventh International Joint Conference on Artificial Intelligence*, 1402-1407.

[DeJong and Mooney 1986] DeJong, G., and Mooney, R. (1986) Explanation-Based Learning: An Alternative View. *Machine Learning* 1:145-176. Kluwer Academic Publishers.

[de Kleer and Brown 1982] de Kleer, J. and Brown, J.S. (1982) Assumptions and Ambiguities in Mechanistic Mental Models. Xerox Technical Report, Cognitive and Instructional Series CIS-9, Corporate Accession P8200028.

[de Kleer and Brown 1984] de Kleer, J. and Brown, J.S. (1984) A Qualitative Physics Based on Confluences. *Artificial Intelligence*, 24:7-83

[Dietterich 1986] Dietterich, T.G. (1986) Learning at the Knowledge Level. *Machine Learning* 1, 287-316.

[Doyle 1979] Doyle, J. (1979) A Truth Maintainence System, *Artificial Intelligence* 12(3).

[Fikes and Nilsson 1971] Fikes, R.E., and Nilsson, N.J. (1971) STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189-208.

[Fisher and Pazzani 1991] Fisher, D. and Pazzani, M. (1991) Computational Models of Concept Learning. In D.H. Fisher, M.J. Pazzani, and P. Langley (eds.): Concept Formation: Knowledge and Experience in Unsupervised Learning. San Mateo, CA: Morgan Kaufmann.

[Flavell 1971] Flavell, J.H. (1971) First discussant's comments: What is memory development the development of? Human Development 14:272-278.

[Forbus 1984] Forbus, K.D. (1984) Qualitative Process Theory.*Artificial Intelligence*, 24:85-168.

[Forbus 1993] Forbus, K.D. (1993) Qualitative Reasoning about Function: A Progress Report. Working Notes of Reasoning About Function Workshop, AAAI 93, July 11-15, 1993, Washington, D.C. pp. 31-36 (Available as a Technical Report from AAAI).

[Franke 1991] Franke, D.W. (1991) Deriving and Using Descriptions of Purpose, In *IEEE Expert* April.

[Freed *et al.* 1992] Freed, M., Krulwich, B., Birnbaum, L., and Collins, G. (1992) Reasoning about performance intentions. Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, July 29-August 1, 1992, Bloomington, Indiana. pp. 7-12. Hillsdale, NJ: Lawrence Erlbaum Associates.

[Friedland 1979] Friedland, P.E. (1979) Knowledge-based experiment design in molecular genetics. Rep No. 79-771, Computer Science Dept., Stanford University. (Doctoral Dissertation.)

[Gentner 1989] Gentner, D. (1989) In *Similarity and Analogical Reasoning* S. Vosniadou and A. Ortony (eds) London: Cambridge University Press.

[Gero *et al.* 1991] Gero, J.S., Tham, K.W., and Lee, H.S. (1991) Behaviour: A Link between Function and Structure in Design. Intelligent CAD'91 Preprints, IFIP, eds. D.C. Brown, H. Yoshikawa amd M. Waldron, pp. 201-230. Columbus, Ohio.

[Gibson 1979] Gibson, J.J. (1979) The ecological approach to visual perception. Boston : Houghton Mifflin.

[Goel 1989] Goel, A. (1989) Integration of Case-Based Reasoning and Model-Based Reasoning for Adaptive Design Problem Solving. Ph.D. Dissertation, Dept. of Computer and Information Science, The Ohio State University.

[Goel and Chandrasekaran 1989] Goel, A. and Chandrasekaran, B. (1989) Functional Representation of Designs and Redesign Problem Solving, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1388-1394.

[Goel and Bylander 1989] Goel A., and Bylander, T. (1989) Computational Feasibility of Structured Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(12):1312-1316, IEEE Press.

[Goel and Chandrasekaran 1990] Goel, A. and Chandrasekaran, B. (1990) A Task Structure for Case-Based Design. In *Proceedings of the 1990 IEEE International Conference on Systems, Man, and Cybernetics*, Los Angeles, California, 587-592, IEEE Systems, Man, and Cybernetics Society Press.

[Goel 1991a] Goel, A. (1991) A Model-Based Approach to Case Adaptation, *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, 143-148.

[Goel 1991b] Goel, A. (1991) Model Revision: A Theory of Incremental Model Learning, *Proceedings of the Eighth International Workshop on Machine Learning*, 605-609.

[Goel *et al.* 1991] Goel, A., Callantine, T., Shankar, M., and Chandrasekaran, B. (1991) Representation, Organization, and Use of Topographic Models of Physical Spaces for Route Planning. In *Proceedings of the Seventh IEEE Conference on AI Applications* pp. 308-314, IEEE Computer Society Press (1991)

[Goel and Callantine 1992] Goel, A. and Callantine, T. (1992) An Experience-Based Approach to Navigational Path Planning. In Proceedings of the IEEE/RSJ International Conference on Robotics and Systems, Rayleigh, North Carolina, July 7-10, Volume II, pp. 705-710. Piscataway, NJ: IEEE Press.

[Goel *et al.* 1993] Goel, A., Gomez, A., Callantine, T., Donnellan, M. and Santamaria, J. (1993) From Models to Cases: where do cases come from and what happens when a case is not available? Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society, Boulder, Colorado, 1993, pp. 474-480. Hillsdale, NJ: Lawrence Erlbaum Associates.

[Goel *et al.* 1995] Goel, A., Ali, K., Donnellan, M., Gomez de Silva Garza, A., Callantine, T. (1995) Integrating Model-Based and Case-Based Methods for Adaptive Navigational Path Planning, To appear in *IEEE Expert*.

[Hamscher and Davis 1987] Hamscher, W., and Davis, R. (1987) Issues in model-based troubleshooting. Memo 893, MIT Artificial Intelligence Laboratory.

[Hammond 1987] Hammond, K. (1987) Explaining and Repairing plans that fail. In *Proceedings ofthe Tenth International Joint Conference on AI*.

[Hammond 1989] Hammond, K. (1989) Case-Based Planning: Viewing Planning as a Memory Task, Boston, MA: Academic Press.

[Hayes 1979] Hayes, P. (1979) Naive Physics Manifesto. *Expert Systems in the Microelectronics Age*, pp. 242-270. Edinburgh, UK: Edinburgh University Press.

[Hayes-Roth and Hayes-Roth 1978] Hayes-Roth, B., and Hayes-Roth, F. (1978) Cognitive processes in planning. Rep. No. R-2366-ONR, Rand Corp., Santa Monica, Calif.

[Hinrichs 1989] Hinrichs, T. (1989) Strategies for Adaptation and Recovery in a Design Problem Solver. *Proceedings Second Case-Based Reasoning Workshop*, 115-118.

[Howe 1992] Howe, A. (1992) Analyzing Failure Recovery to Improve Planner Design. In Proceedings of the Tenth National Confrence on AI, pp. 387-392.

[Huhns and Acosta 1988] Huhns, M., and Acosta, E. (1988) ARGO: A System for Design by Analogy. *IEEE Expert*.

[Hunter 1990] hunter90) Hunter, L. (1990) Planning to learn. In *Proceedings Twelvth Cognitive Science Society Conference*, 26-34.

[Johnson 1993] Johnson, K. (1993) Exploiting a Functional Model of Problem Solving for Error Detection in Tutoring, Ph.D. Dissertation, The Ohio State University.

[Karmiloff-Smith 1979] Karmiloff-Smith, A. (1979) Micro- and macro developmental changes in language acquisition and other representational systems. In Cognitive Science, 3 pp. 91-118.

[Kluwe 1982] Kluwe, R.H. (1982) Cognitive Knowledge and Executive Control: Metacognition. In Animal Mind - Human Mind, ed. D.R. Griffin, pp. 201-224, Berlin, Heidelberg, New York:Springer-Verlag.

[Kluwe and Schiebler 1984] Kluwe, R.H., and Schiebler, K. (1984) Entwicklung executiver prozesse und kognitiver leistungen. In F.E. Weinert and R.H. Kluwe (eds.), Metacognition, Motivation, and Understanding, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Knoblock 1990] Knoblock, G. (1990) Learning Abstraction Hierarchies for problem Solving. In *Proceedings of the Eighth National Conference on AI*, 923:928.

[Kolodner 1987] Kolodner, J. (1987) Capitalizing on failure through case-based inference. In the *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Northvale, NJ: Erlbaum.

[Kolodner and Simpson 1984] Kolodner, J. and Simpson, R. (1984) Experience and problem solving: A framework In the *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, Northvale, NJ: Erlbaum.

[Kolodner and Simpson 1989] Kolodner, J. and Simpson, R. (1989) The MEDIATOR: Analysis of an Early Case-Based Problem Solver. *Cognitive Science* 13(4): 507-549.

[Kolodner 1993] Kolodner, J. (1993) Case-Based Reasoning. San Mateo, CA: Morgan Kaufmann.

[Koton 1988] Koton, P. (1988) Using Experience in Learning and Problem Solving. Ph.D.. Dissertation, Dept. of Computer Science, MIT, 1989.

[Korf 1983] Korf, R.E. (1983) Learning to Solve Problem by Searching for Macro-Operators, Rep. No. CMU-CS-83-138, Computer Science Dept., Carnegie Mellon University.

[Kuipers 1984] Kuipers, B. (1984) Commonsense Reasoning About Causality. *Artificial Intelligence*, 24:169-203.

[Kuipers 1985] Kuipers, B. (1985) Qualitative Simulation in Medical Physiology: A Progress Report, MIT, Laboratory for Computer Science Report MIT/LCS/TM-280.

[Kuipers 1986] Kuipers, B. (1986) Qualitative Simulation. *Artificial Intelligence*, 29:289-338.

[Kuipers and Levitt 1988] Kuipers, B., and Levitt, T. (1988) Navigation and Mapping in Large-Scale Space. *AI Magazine*, 9(2): 25-43.

[Kumar and Upadhyaya 1992] Kumar, A. and Upadhyaya, S. (1992) Framework for function-based Diagnosis. technical Report, State University of New York at Buffalo, Buffalo NY 14260.

[Kuokka 1990] Kuokka, D.R. (1990) The Deliberative Integration of Planning, Execution, and Learning, Rep. No. CMU-CS-90-135, Ph.D. Dissertation, Computer Science Dept., Carnegie Mellon University.

[Laird *et al.* 1986] Laird, J., Rosenbloom, P., and Newell, A. (1986) Chunking in SOAR: The anatomy of a General Learning Mechanism. Dordrecht, The Netherlands: Kluwer Academic Publishers.

[Laird *et al.* 1987] Laird J., Newell, A., and Rosenbloom, P. (1987) SOAR: An Architecture for General Intelligence. Artificial Intelligence, 33:1-64.

[Langley 1980] Langley, P.W. (1980) descriptive discovery processes: experiments in Baconian science. Rep. No. CS-80-121, Ph.D.. Dissertation, Computer Science Dept., Carnegie Mellon University.

[Langley and Simon 1981] Langley P.W., and Simon, H.A. (1981) The Central Role of Learning in Cognition. In J.R. Anderson (ed.) *Cognitive Skills and Their Acquisition.* Hillsdale, N.J: Erlbaum, pp. 361-380.

[Lee *et al.* 1994] Lee, J., Huber, M.J., Durfee, E., Kenny, P. (1994) UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space* CIRFFSS'94.

[Maes 1987] Maes, P. (1987) Computational Reflection. Technical Report 87-2, Free University of Brussels, AI Lab.

[Marcques *et. al* 1992] Marcques, D., Dallemagne, G., Klinker, G., McDermott, J., and Tung D. (1992) Easy Programming: Empowering people to build their own applications. IEEE Expert, June 1992, 16-29.

[Marr 1982] Marr, D. (1982) *Vision: A Computational investigation into the Human Representation and Processing of Visual Information*, W.H. Freeman, San Francisco.

[Martin 1992] Martin, J.D. (1992) Direct and indirect transfer : explorations in concept formation. Ph.D. Dissertation, Georgia Institute of Technology.

[McDermott 1978] McDermott, D. (1978) Planning and Acting, *Cognitive Science* vol. 2.

[McDermott 1981] McDermott, D. (1981) Artificial Intelligence Meets Natural Stupidity, in Mind Design: Philosophy, Psychology, Artificial Intelligence, J. Haugeland (ed.), Cambridge, MA: MIT Press.

[McDermott and Davis 1984] McDermott, D., and Davis E. (1984) Planning Routes through Uncertain territory. *AI* 22:107:156.

[McDermott 1988] McDermott, J. (1988) Preliminary steps toward a taxonomy of problem-solving methods, In Automating Knowledge Acquisition for Expert Systems, ed. S. Marcus, pp. 225-266. Dordrecht, The Netherlands:Kluwer Academic Publishers.

[Michalski 1991] Michalski, R.S. (1991) Inferential learning theory as a basis for multi-strategy adaptive learning. In R.S. Michalski and G. Tecuci (eds.): Proceedings of the First International Workshop on Multistrategy Learning, 3-18, Harpers Ferry, WV.

[Minsky 1963] Minsky, M. (1963) Steps Towards Artificial Intelligence. In Feigenbaum and Feldman (eds): Computers and Thought, McGraw-Hill, New York.

[Minsky 1968] Minsky, M. (1968) The Mind, Matter, and Models paper. In Minsky, M. (ed.), *Semantic Information Processing*, 227-270. Cambridge, MA: MIT Press.

[Minton 1990] Minton, S. (1990) Qualitative results concerning the utility of explanation-based learning. Artificial Intelligence 42:363-392.

[Mischel 1981] Mischel, W. (1981) Metacognition and the rules of delay. In J.H. Flavell and L. Ross (eds.) *Social Cognitive Development* 240-271, Cambridge, Cambridge University Press.

[Mitchell *et al.* 1981] Mitchell, T.M., Utgoff, P.E. Nudel, B., and Banerji, R.B. (1981) Learning problem-solving heuristics through practice. In *Proceedings of the Seventh International Joint Conference on AI* 127-134.

[Mitchell *et al.* 1986] Mitchell, T.M., Keller, R.M., and Kedar-Cabelli T. (1986) Explanation-based generalization: a unifying view. *Machine learning* 1:47-80, Kluwer Academic Publishers.

[Mitchell *et al.* 1989] Mitchell, T.M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., and Schlimmer, J. (1989) Theo: A Framework for Self-Improving Systems. In K. VanLehn (ed.): Architectures for Intelligence, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Mostow 1990] Mostow, J. (1990) Design by Derivational Analogy: Issues in the Automated Replay of Design Plans. *Artificial Intelligence*

[Newell and Simon 1963a] Newell A., and Simon, H.A. (1963) GPS: A program that simulates human thought, in E.A Feigenbaum and J. Feldman (eds.) *Computers and Thought* 279-293, R. Oldenbourg KG.

[Newell and Simon 1963b] Newell A., and Simon, H.A. (1963) The Theory of Human Problem Solving. Reprinted in Readings in Cognitive Science, Collins and Smith (eds.).

[Newell and Simon 1972] Newell, A. and Simon, H.A. (1972) *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

[Newell 1982] Newell, A. (1982) The Knowledge level *AI Journal* 19(2):87-127.

[van Nijnatten 1993] van Nijnatten, C., (1993) KADskit, KADS modelling using KREST. AI memo 93-06. Free University of Brussels.

[Pazzani 1988] Pazzani, M. (1988) Learning Causal Relationships: An Integration of Empirical and Explanation-Based Learning Methods, University of California, Los Angeles (UCLA), Report UCLA-AI-88-10.

[Perez 1994] Perez, M.A. (1994) The goal is to produce better plans. In the Working Notes of the Goal-Driven Learning Symposium, AAAI Spring Symposium series, March 21-23, Stanford University.

[Piaget 1971] Piaget, J. (1971) Biology and Knowledge. University of Chicago Press.

[Piaget 1976] Piaget, J. (1976) The grasp of consciousness: Action and concept in the young child. Cambridge MA: Harvard University Press.

[Pirolli and Recker 1992] Pirolli, P., and Recker, M. (1992) Learning Strategies and Transfer in the Domain of Programming, In Cognition and Instruction.

[Plaza and Arcos 1993] Plaza, E., and Arcos, J.L. (1993) Reflection and Analogy in Memory-based Learning. In R. S. Michalski and G. Tecuci (eds.): Proceedings of the Second International Workshop on Multistrategy Learning.

[Price and Hunt 1992] Price, C.J. and Hunt, J.E. (1992) Automating FMEA through multiple models, In *Research and Development in Expert Systems VIII*, I. Graham (editor), pp 25-39.

[Hunt *et al.* 1993] Hunt, J.E., Price, C.J., Lee, M.H. (1993) Automating the FMEA process. Intelligent Systems Engineering (2)2 pp. 119-132

[Quinlan 1986] Quinlan, J.R., (1986) Induction of decision trees. *MAchine Learning* 1, 86-106.

[Quinlan 1986] Quinlan, J.R., (1986) The effect of Noise in Concept learning. In *Machine Learning: An Artificial Intelligence Approach (Vol II)*, R. S. Michalski and J. G. Carbonell and T. M. Mitchell (editors), San Mateo, CA: Morgan Kaufman.

[Ram 1989] Ram, A. (1989) *Question driven understanding: An integrated theory of Story understanding, memory and learning*. Ph.D. thesis, Yale University, department of Computer Science, New Haven, CT, Research Report# 710.

[Ram 1991] Ram, A. (1991) A theory of questions and question asking. *The Journal of the Learning Sciences*, 1(3&4). In press.

[Ram and Hunter 1992] Ram, A., and Hunter, L. (1992) The use of explicit goals for knowledge to guide inference and *Journal of Applied Intelligence* 2(1):47-73, 1992, Also available as Technical Report GIT-CC-92-03, College of Computing, Georgia Inst. of Technology.

[Ram *et al.* 1993] Ram, A., Narayanan, S., and Cox M.T. (1993) Learning to Troubleshoot: Multistrategy Learning of Diagnostic Knowledge for a Real-World Problem Solving Task. To appear in *Cognitive Science*, Also available as a GeorgiaTech Technical Report GIT-CC-93/67.

[Ram and Cox 1994] Ram, A. and Cox M.T. (1994) Introspective Reasoning Using Meta-Explanations for Multistrategy Learning. In Machine Learning: A Multistrategy Approach IV. eds. R.S. Michalski and G. Tecuci, pp. 349-377. San Mateo, CA: Morgan Kaufmann,

[Rasmussen 1985] Rasmussen, J. (1985) The role of hierarchical knowledge representation in decision making and system management. *IEEE Transcations on Systems, Man, and Cybernetics*, SMC 15(2), 234-243.

[Rasmussen 1986] Rasmussen, J. (1986) *Information Processing and and human-machine interaction: an approach to cognitive engineering* North-Holland.

[Rasmussen 1988] Rasmussen, J. (1988) Models for design of computer integrated manufacturing systems. In Ergonomics of Hybrid Automated Systems I. Proceedings of the First International Conference on Ergonomics of Advanced Manufacturing and Hybrid Automated Systems, Louisville, KY, USA 15-18 Aug.

[Ross 1986] Ross, B.H. (1986) Remindings in Learning: Objects and Tools. In S. Bosniadou, and A. Ortony (eds.): Similarity, analogy and thought. New York: Cambridge Univ. Press.

[Sacerdoti 1974] Sacerdoti, E.D. (1974) Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115-135.

[Sacerdoti 1975] Sacerdoti, E.D. (1975) A structure for plans and behavior. Tech Note 109, AI Center, SRI International, Inc., Menlo Park, Calif. (Doctoral Dissertation.)

[Samuel 1959] Samuel, A. (1959) Some studies in machine learning using the game of checkers, IBM Journal of R&D Reprinted in Computers and Thought, Feigenbaum and Feldman (eds.), (1963).

[Schlimmer 1990] Schlimmer, J.C. (1990) Decompiling problem-Solving Experience to Elucidate Representational Distinctions. In *Change of Representation and Inductive Bias*. Dordrecht, The Netherlands:Kluwer Academic Publishers.

[Shortliffe 1976] Shortliffe, E.H. (1976) *Computer-based medical consultations: MYCIN*. New York, NY: American Elsevier.

[Sembugamoorthy and Chandrasekaran 1986] Sembugamoorthy, V. and Chandrasekaran, B. (1986) Functional Representation of Devices and Compilation of Diagnostic Problem-Solving Systems. In Experience, Memory, and Reasoning, eds. J. Kolodner and C. Riesbeck, pp. 47-73. Hillsdale, NJ: Lawrence Erlbaum Associates.

[Simon 1972] Simon, H.A. (1972) Complexity and the representation of patterned sequences of symbols, *Psychological review*, 79:369-382.

[Simon 1981] Simon, H.A. (1981) The Sciences of the Artificial. Cambridge, MA: MIT Press.

[Simon 1984] Simon, H.A. (1984) Why Should Machines Learn? *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski and J. G. Carbonell and T. M. Mitchell (editors), Springer-Verlag, Berlin, Heidelberg, pp. 25-37.

[Simmons 1988] Simmons, R.G. (1988) Combining Associational Causal Reasoning to Solve Interpretation and Planning Problems, MIT Report 1048.

[Sussman 1975] Sussman, J.G. (1975) A Computational Model of Skill Acquisition, American Elsevier, New York.

[Steels 1990] Steels, L. (1990) Components of Expertise. AI Magazine 11:30-4.

[Stefik 1980] Stefik, M. (1980) Planning with Constraints. Rep No. 80-764, Computer Science Dept., Stanford University. (Doctoral Dissertation.)

[Stefik 1981] Stefik, M. (1981) Planning and Meta-Planning (MOLGEN: Part 2), In *Artificial Intelligence*, 16:141-169.

[Sticklen and Chandrasekaran 1989] Sticklen, J. and Chandrasekaran, B. (1989) Integrating classification-based compiled level reasoning with function-based deep level reasoning. In *Causal AI Models: Steps towards Applications*, W. Horn (editor), Hemisphere Publishing Corporation, pp 191-220.

[Stroulia 1992] Stroulia, E. (1992) Towards a Functional Model of Reflective Learning, Tech. Report, GIT-CC-92-56.

[Stroulia and Goel 1992a] Stroulia, E. and Goel, A. (1992) Generic Teleological Mechanisms and their Use in Case Adaptation. In the *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pp. 319-324. (1992)

[Stroulia *et al.* 1992] Stroulia, E., Shankar, M., Goel, A. and Penberthy, L., (1992) A Model-Based Approach to Blame Assignment in Design. In Proceedings of the Second International Conference on AI in Design, ed. J.S. Gero, pp. 519-537. Dordrecht, The Netherlands:Kluwer Academic Publishers.

[Stroulia and Goel 1992b] Stroulia, E. and Goel, A. (1992) An Architecture for Incremental Self-Adaptation, In Proceedings of Machine Learning Workshop on Computational Architectures for Supporting Machine Learning and Knowldege Acquisition, July 4, 1992, Aberdeen Scotland.

[Stroulia and Goel 1993] Stroulia, E., and Goel, A. (1993) Functional Representation and Reasoning for Reflective Systems. Applied Artificial Intelligence: An International Journal (to appear).

[Stroulia and Goel 1994a] Stroulia, E., and Goel, A. (1994) Trade-Offs in Acquiring Problem-Decomposition Knowledge: Some Experiments with the Principle of Locality. In the Proceedings of the Eighth Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff Canada.

[Stroulia and Goel 1994b] Stroulia, E., and Goel, A. (1994) Learning Problem-Solving Concepts by Reflecting on Problem Solving. To appear in the Proceedings of the ECML-94, 7th European Conference on Machine Learning, 5-8 April 1994, Catania, Sicily.

[Sycara and Navinchandra 1989] Sycara, K., and Navinchandra, D. (1989) A Process Model of Experience-Based Design *Proceedings Eleventh Cognitive Science Society Conference*, 283-290.

[Sycara and Navinchandra 1989] Sycara, K. and Navinchandra D. (1989) Integrating Case-Based Reasoning and Qualitative Reasoning in Engineering Design". In Artificial Intelligence in Engineering Design, J. Gero (Ed.), Computational Mechanics Publications, U.K., July 1989, Computational Mechanics Publications, Southampton U.K. and Boston U.S.A and Springer-Verlag, Berlin and Heidelberg, Germany, pp. 232-250.

[Tadjer 1993] Tadjer, V., (1993) Using the KREST Workbench for the Development of Reusable Knowledge Systems for Process Control: A limited case study for management of alarm situations. AI memo 93-10. Free University of Brussels.

[Tate 1975a] Tate, A. (1975) Interacting goals and their use. In *Proceedings of the Fourth International Joint Conference on AI*, pp. 215-218.

[Turner 1989] Turner, R. (1989) A schema-based model of adaptive problem solving. Ph.D. Dissertation, Georgia Institute of Technology, GIT-CC-92/43.

[Umeda *et al.* 1990] Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H. (1990) Function, Behaviour, and Structure. In Gero (ed.): *Applications of Artificial Intelligence in Engineering, vol 1, Design, Proceedings of the Fifth International Conference*, pp. 177-193. Berlin: Springer-Verlag.

[Van de Velde 1988] Van de Velde, W. (1988) Learning from experience. Ph.D. Dissertation, Free University of Brussels.

[Veloso 1992] Veloso, M. (1992) Learning by Analogical Reasoning in general Problem Solving. Ph.D.. Dissertation, Computer Science Dept., Carnegie Mellon University.

[Waterman 1968] Waterman, D.A. (1968) Machine learning of heuristics. Ph.D. Dissertation, Computer Science Department, Stanford University, Tech Report # STAN-CS-68-118.

[Weintraub 1991] Weintraub, M. (1991) An Explanation-Based Approach to Assigning Credit, Ph.D. Dissertation, The Ohio State University.

[Weinart 1987] Weinart, F.E. (1987), Introduction and Overview: Metacognition and Motivation as Determinants of Effective Learning and Understanding, in F. E. Weinert and R. H. Kluwe (eds.), Metacognition, Motivation, and Understanding, Hillsdale, NJ: Lawrence Erlbaum Associates.

[Wielinga and Breuker 1986] Wielinga, B.J., and Breuker, J.A. (1986) Models of expertise, In *Proceedings Seventh European Conference on Artificial Intelligence*, 306-318, Brighton.

[Wielinga *et al.* 1992] Wielinga, B.J., Schreiber, A.Th., and Breuker, J.A. (1992) KADS: A modelling approach to knowledge engineering. Knowledge Acquisition, 4(1) pp. 5-53. Special issue "The KADS approach to knowledge engineering".

[Wilensky 1983] Wilensky, R. (1983) *Planning and Understanding, A Computational Approach to Human Reasoning*. Addison Wesley.

[Wilensky 1984] Wilensky, R. (1984) Meta-Planning:Representing and using knowledge about planning in problem solving and natural language understanding, *Cognitive Science* 5:197-234.

[Wilkins *et al.* 1987] Wilkins, D.C., Clancey, W.J., and Buchanan, B.G. (1987) Knowledge Base Refinement by Monitoring Abstract Control Knowledge, Stanford University, Report STAN-CS-87-1182; also KSL-87-01.

[Winston 1975] Winston, P. (1975) Learning Structural Descriptions from Examples, in The Psychology of Computer Vision, P.H. Winston (ed.).

[Winston 1982] Winston, P. (1982) Learning New Principles from Precedents and Exercises. *Artificial Intelligence* 19.

[Wisniewski and Medin 1991] Wisniewski, E.J., and Medin, D.L. (1991) Harpoons and Long Sticks: The Interaction of Theory and Similarity in Rule Induction, In D.H.Fisher, M.J.Pazzani, and P.W.Langley (eds.) *Concept Formation: Knowledge and Experience in Unsupervised Learning*, San Mateo, CA: Morgan Kaufmann.

[Wisniewski and Medin 1994] Wisniewski E.J., and Medin D.L. (1994) On the interaction of theory and data in concept learning. *Cognitive Science* v. 18, pp. 221.

VITA

Eleni Stroulia was born on the 29th of January 1967, in Larissa, Greece (father's name Alexandros Stroulias, mother's maiden name Asimina Koutsimani). She received her B.S. degree in Computer Engineering and Informatics from the University of Patras, Greece, in 1989. She attended the graduate program at the College of Computing of the Georgia Institute of Technology from September 1989 to December 1994. During the summer of 1991, she worked as a summer intern for AT&T Global Information Solutions (then NCR Corporation), Human Interface Technology Center, in Atlanta, Georgia. In March 1991 she received her M.S. in Computer Science from the Georgia Institute of Technology and in December 1994 she completed her Ph.D. in Computer Science, with specialization in Artificial Intelligence and Cognitive Science. Her research interests include knowledge-based systems, learning, design, functional reasoning, and autonomous agents.